A Pragmatic Approach to Understand Reflective Programming: Building ObjVlisp a Minimal, Uniform and Reflective Object-Oriented Language

Part of the 2003/2004 Wintersemester

Dr. Stéphane Ducasse
Alexandre Bergel
Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland
{ducasse, bergel}@iam.unibe.ch
http://www.iam.unibe.ch/∼scg/

November 7, 2003

1 Objectives

During the lecture you saw the main points of the ObjVLisp model, now you will implement it. This implementation aims at providing you a concrete understanding of the concepts presented in the lecture. Here follow some of the points you can get from doing such an implementation.

- What is a possible example of the structure of an object?
- What is the allocation and the initialization of an object?
- 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?
- What is the notion of meta-programming?

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 Files Provided

The parcel[1] named ObjVLisp-ForStudent contains already all the classes, the method categories and the method signature that you need to implement. Moreover extra functionality is provided like a dedicated inspector and some necessary but not interesting functionality.

It also contains 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.

To install ObjVLisp: you need to download the files ObjVLisp-ForStudent.pcl and ObjVLisp-ForStudent.pst. The former contains all the binary of the code, and the latter all the source code. Just load the .pcl file, the source file will be load automatically.

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 class name 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. ObjPoint is the name of an objClass. #(#ObjClass #ObjPoint #ObjObject #(class x y) #(:x :y) nil)) is an objClass.

2.3 Implementation Choices

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 contains an easy way like in Lisp[2] to define macroes, we will use as much as possible the existing classes to avoid extra syntaxic problems.

Let's look at the problem. We could have implemented ObjVLisp functionality at the class level of a self-contained class named Obj (inheriting only from Object). However, using the ObjVlisp primitive (a Smalltalk method) objInstanceVariableValue:for:

[1] a parcel is a unit of deployment in Visualworks

[2] This is not true, some methods like ifTrue: are expanded at compile-time.

that given object, an instance variable name returns the value of the instance variable would have led to code like the following:

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

As we chose to represent any ObjVLisp object by an array, we chose to define the ObjVLisp functionality in a subclass of Array named Obj. With this choice, inheritance is considered as an implementation relationship and not as a subtype relationship between the two classes because the type of an ObjVLisp object is not Array. The same functionality as above is more natural and readable:

```
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. We defined the following methods to store and access defined classes.

- declareClass: anObjClass stores an ObjClass in the defined 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
```

This kind of code makes the writing of code heavy. To ease the writing of the code we use a trick that will be explained later in detail during the lecture. we trap messages not understood sent to Obj and look into the defined class dictionary. This same code is then written

```
Obj ObjPoint objClass
```

3 Structure and Primitives

The first issue is how to represent the 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 hardcod the class structure.

Your job: Check that the class Obj exists and inherits from Array. Note that as Array is a class with indexed variables, we used the message variableSubclass:.. instead of subclass:.. for its creation.

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 of the system. Given an array (in the following we used the terms array for talking about of instances of the class Obj) a class as 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)) meaning that ObjPoint is named ObjPoint, is an instance of ObjClass, inherits from ObjObject, has three instance variables, two initialization keywords and a uninitialized method dictionary. To access this structure we define some primitives.

To help you to implement ObjVLisp we provide you an adapted inspector dedicaced to inspecting ObjVLisp objects. You can invoke this inpector in the following ways:

```
| 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 method primitiveStructure of the class ObjTest gives some examples of structure accesses. Implement the primitives that are missing to run the test (ObjTest selector: #testPrimitiveStructure) run. Note that array starts at 1 in Smalltalk. Below is the list of the primitives 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 an information identifying its class (not directly its class for recursive problem). In this simple implementation when an instance is created its first instance variable contains a symbol that is the name of its class.

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, defines the primitive objClass in the category ‘objectstructure primitive’ that sent to an objInstance 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 sent to an objClass with aSymbol 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 sent to an instance with a symbol returns the offset of the instance variable and (b) the method valueOfInstanceVariable: aSymbol that sent to an instance with a symbol 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.

  Evaluate: (ObjTest selector: #testIVOffsetAndvalue) run.

4 Allocation and Initialisation

The creation of an object is the composition of two elementary operations: its *allocation* and its *initialization*. We now will 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 a instance method that given a newly allocated instance and a list of initialization arguments fill the objects.

4.1 Allocation

Your job: In category 'instance allocation' implement the primitive called allocateAnInstance that sent to a *objClass* returns a new instance whose instance variable values are nil. 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 now have to implement some functionality to support keywords. However as this is not really interesting that you lose time we provide 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: Implement in the category 'instance initialization' the primitive initializeUsing: anArray that sent an object with an initargs-list returns an initialized object. The code of this method is given below. Evaluate: (ObjTest selector: #testInitialize) run.

```
Obj >>initializeUsing: anAlternatedArray
  "Returns the receiver an ObjObject initialized according
```

```
to the directives given by anAlternateArray"

    | ivValues |
    ivValues := self  returnValuesFrom: anAlternatedArray
                    followingSchema: self objClass objKeywords.
    ^ivValues startingAt: 1
          replaceElementsIn: self
          from: 2
          to: ivValues size + 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 variable as the union 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' implement the primitive computeNewIVFrom: superIVOrdCol with: localIVOrdCol that given two ordered collections of symbols 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.
Technical Note: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 you wrote in section **??**. That's why your implementation of the primitive computeNewIVFrom:with: should take care of that aspect.

6 Method Management

A class stores the behavior shared by all its instances into a method dictionary. We implement methods by associating a symbol with a Smalltalk block. The following code presents a possible x method defined on the objClass ObjPoint and sent to an objInstance of this class [:objself | objself binarySend: #getIV with: #x].

In this implementation we do not provide the ability to access directly instance variables of the class in which the method is defined. The programmer has to use accessors or the setIV and getIV objMethods defined on ObjObject. We provide you all the primitives that deals with method definition.

Your job: In the category 'method management' look at the methods addMethod: aSelector withBody: aBlock, removeMethod: aSelector, bodyOfMethod: aSelector and doesUnderstand: aSelector
Evaluate: (ObjTest selector: #testMethodManagement) run.

7 Message Passing and Dynamic Lookup

Sending a message is the result of the composition of the method lookup and the method application. Thus 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 next subsection clarify the why and how of ClassesImplementingLookupMethod.

7.1 Stack of Class

7.1.1 The Problem

Before you implement the lookup, a few explanation need to be said regarding the class where the method lookup start. Whenever a message is sent to an object, the class where the lookup has to start is given by this object: It corresponds to its class. But things are a 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 send a message to super). The problem is this class need to be referenced, and that is what ClassesImplementingLookupMethod is doing.

In Java and Smalltalk there is no such problem because there is a compilation phase. So during the compilation of a class, whenever super is encountered, the compiler knows what is the super class (it correspond to the superclass of the class being compiled).

In the case of ObjVLisp the compiler does not have such information, so the class of the method execute has to be keep. A method, when executed, has a reference to the receiver, but has not a reference to the class which define this method. And it is necessary to know it when super is referenced. This information has to be computed at run-time, whereas 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.

7.1.2 Maintaining a Stack

The method Obj class>>initializeStack set the shared variable ClassesImplementingLookupMethod to reference an empty ordered collection. This variable represents the stack. During the lookup process, when a method is found, the class which define 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`

7.2 Method Lookup

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

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 that 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.
Evaluate: (ObjTest selector: #testMethodLookup) run.

7.3 Complementary Self and Super Sends

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 propose you to implement three different primitives for message passing: one for unary messages unarySuper:, one for binary messages binarySuper: and one for n-ary messages super:withArguments:.

Your job: Look at three primitives below and implement in the category 'message passing' the primitives unarySuper: selector, binarySuper: selector with: argument and super: selector withArguments: anArray.

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

Obj >>binarySend: selector with: argument
| ans |
ans := (self objClass lookup: selector for: self) value: self value: argument.
ClassesImplementingLookupMethod removeLast.
^ ans

Obj >>send: selector withArguments: arguments
| ans |
ans := (self objClass lookup: selector for: self) valueWithArguments: (Ar-
ray with: self) , arguments.
ClassesImplementingLookupMethod removeLast.
^ ans
```

Evaluate: (ObjTest selector: #testMethodSelfSend) run.

8 Bootstrapping the system

Now you have implemented all the functionality 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.

```
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 define another set of tests in the class ObjTestBootstrap.

8.1 Manually creating 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 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 all the primitives that compose it.

Your job: At the class level in the category 'bootstrap objClass manual' 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 provide 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]
```

- 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.
- the primitive manuallyCreateObjClass as shown below:

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

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

Evaluate: (ObjTestBootstrap selector: #testManuallyCreateObjClassAllocate) run.

Your job: Read carefully the following remarks below and the code.

- In the objMethod manualObjClassStructure 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.
- Note the method #initialize is a class method. 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: 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

- the primitive objObjectStructure that creates the ObjObject by invoking the new message to the class ObjClass as shown hereafter:

```
Obj class>>objObjectStructure

    ^ 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.

Evaluate: (ObjTestBootstrap selector: #testCreateObjObjectStructure) run.

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.

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 stccreateObjClass is responsible to create the final class ObjClass. So you will implement it and define all the primitive it needs.

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

    | objClass |
    objClass := self objClassStructure.
    self defineAllocateMethodIn: objClass.
    self defineNewMethodIn: objClass.
    self defineInitializeMethodIn: objClass.
    objClass addMethod: #isMetaclass
        withBody: [:class | class objIVs includes: #superclass].
        "an object is a class if its class is a metaclass. cool"
    objClass addMethod: #isClass
        withBody: [:class | class objClass unarySend: #isMetaclass].
    ^objClass
```

Implement

- 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.

```
Obj class>> objClassStructure

  ^ObjClass send: #new
        withArguments:
            #(#(#name: #ObjClass #iv: #(#name #superclass #iv #key-
words #methodDict) #superclass: #ObjObject))
```

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
```

Implement

- the objMethod isClass that returns true.
- the objMethod isMetaClass that returns true.
- the primitive defineInitializeMethodIn: anObjClass that adds the objMethod initialize to the objClass passed as argument. The objMethod initialize takes the receiver and an initargs-list and initializes the receiver according to the specification given by the initargs-list.

```
Obj class>>defineInitializeMethodIn: anObjClass
anObjClass addMethod: #initialize
      withBody: [:aclass :initArray |
                  aclass binarySuper: #initialize with: initArray.
                     aclass objIVs: (aclass computeNewIVFrom: (Obj give-
ClassNamed: aclass objSuperclassId) objIVs
                                          with: aclass objIVs).
                  aclass computeAndSetKeywords.
                  aclass objMethodDict: IdentityDictionary new.
                  Obj declareClass: aclass.
                  aclass].
```

- the code of the primitive createObjClass is then the following one.

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

      | objClass |
      objClass := self objClassStructure.
      self defineAllocateMethodIn: objClass.
```

```
        self defineNewMethodIn: objClass.
        self defineInitializeMethodIn: objClass.
        objClass addMethod: #isMetaclass
              withBody: [:class | class objIVs includes: #superclass].
              "an object is a class if is class is a metaclass. cool"
        objClass addMethod: #isClass
              withBody: [:class | class objClass unarySend: #isMetaclass].
        ^objClass
```

Evaluate: (ObjTest 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.

```
| pointClass aPoint |
pointClass := ObjClass send: #new
                withArguments: #((#name: #ObjPoint
                                  #iv: #(#x #y)
                                  #superclass: #ObjObject)).
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.

```
ObjPoint addMethod: #x
      withBody: [:objself | objself binarySend: #getIV with: #x].

ObjPoint addMethod: #x:
          withBody: [:objself :val | objself send: #setIV withArguments: (Ar-
ray with: #x with: val)].

ObjPoint addMethod: #display
      withBody:  [:objself |
                    Transcript cr; show: 'aPoint with x = '.
                    Transcript show: (objself unarySend: #x) printString; cr].
```

Then test these new functionality.

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

9.2 ObjColoredPoint

Define the class ObjColored.

```
| coloredPointClass aColoredPoint |
ObjClass send: #new
      withArguments: #((#name: #ObjColoredPoint
                  #iv: #(#color)
                  #superclass: #ObjPoint)).
```

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.

```
coloredPointClass
    addMethod: #color
    withBody: [:objself | objself binarySend: #getIV with: #color].
```

```
coloredPointClass
    addMethod: #color:
    withBody: [:objself :val | objself send: #setIV
                                withArguments: (Array with: #color with: val)].

coloredPointClass
    addMethod: #display
    withBody: [:objself |
                objself unarySuper: #display.
                Transcript cr;  show: ' with Color = '.
                Transcript show: (objself unarySend: #color) printString; cr].
aColoredPoint unarySend: #x.
aColoredPoint unarySend: #color.
aColoredPoint unarySend: #display
```

---

10 First User Metaclasses: ObjAbstract and ObjSet

Now implement the metaclass ObjAbstract that defines instances (classes) that are abstract i.e. that cannot create instances.

```
| abstractClass   |
abstractClass := #ObjClass
                send: #new
                withArguments: #(#(#name: #ObjAbstractClass
                            #iv: #()
                            #superclass: #ObjClass)).
abstractClass addMethod: #new
          withBody: [:class :initArray |
                    class error: ' the class ',class objName asString,' is abstract'].
```

Then the following show you how to use it.

```
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 ot because this is a metaclass.

Your job: Implement the metaclass ObjSet the metaclass that defines as class behavior the fact that a class knows its instances.

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:.