

# Un kernel object réflexif simplifié

Stéphane Ducasse

November 22, 2021

Copyright 2017 by Stéphane Ducasse.

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

You are **free**:

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

Under the following conditions:

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

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

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

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



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

# Contents

<b>Illustrations</b>	<b>iii</b>
<b>1 Un noyau minimal réflexif basé sur les classes</b>	<b>3</b>
1.1 L'inspiration d'ObjVlisp	3
1.2 Les six postulats d'ObjVLisp	4
1.3 Aperçu du noyau	5
1.4 Instances	7
1.5 Comprendre les métaclasse	8
1.6 Structure de l'instance	9
1.7 A propos du comportement	10
1.8 La classe en tant qu'objet	10
1.9 Envoi d'un message	13
1.10 Héritage	15
1.11 Objet : définition du comportement minimal de tout objet	16
1.12 Héritage et instanciation ensemble	17
1.13 Révision de la sémantique du self et du super	17
1.14 Création d'objets	21
1.15 Création d'instances de la classe Point	21
1.16 Création de la classe Point instance de Class	22
1.17 La classe Class	25
1.18 Définition d'une nouvelle métaclasse	26
1.19 À propos de l'état des classes	28
1.20 À propos du 6e postulat	30
1.21 Conclusion	31
<b>2 Construction d'un noyau minimal réflexif basé sur des classes</b>	<b>33</b>
2.1 Objectifs	33
2.2 Préparation	34
2.3 Conventions de nommage	35
2.4 Hériter de la classe Array	35
2.5 Faciliter l'accès à la classe objclass	36
2.6 Structure et primitives	37
2.7 Structure d'une classe	37
2.8 Trouver la classe d'un objet	39
2.9 Accès aux valeurs des variables d'instance des objets	39
2.10 Allocation et initialisation des objets	41

2.11	Primitives de mots-clés . . . . .	42
2.12	Initialisation des objets . . . . .	43
2.13	Héritage statique des variables d'instance . . . . .	43
2.14	Gestion des méthodes . . . . .	44
2.15	Passage de messages et recherche dynamique . . . . .	46
2.16	Recherche de méthode . . . . .	47
<b>3</b>	<b>Gestion de super</b>	<b>49</b>
3.1	Représentation de super . . . . .	50
3.2	Gestion des messages non compris . . . . .	51
<b>4</b>	<b>Amorçage du système</b>	<b>53</b>
4.1	Création manuelle de la classe ObjClass . . . . .	54
4.2	Création d'un ObjObject . . . . .	56
4.3	Création de la classe ObjClass . . . . .	57
4.4	Premières classes d'utilisateurs : ObjPoint . . . . .	60
4.5	Premières classes d'utilisateurs : ObjColoredPoint . . . . .	61
4.6	Une métaclasse pour les premiers utilisateurs : ObjAbstract . . . . .	62
4.7	Nouvelles fonctionnalités que vous pouvez implémenter . . . . .	62
<b>5</b>	<b>Définitions choisies</b>	<b>65</b>
	<b>Bibliography</b>	<b>67</b>

# Illustrations

1-1	Le noyau ObjVlisp : un noyau minimal basé sur les classes. . . . .	6
1-2	Le noyau avec des métaclassees spécialisées. . . . .	6
1-3	Comprendre les métaclassees à l'aide du passage de messages. . . . .	7
1-4	Chaîne d'instanciation : les classes sont aussi des objets. . . . .	8
1-5	Tout est un objet. Les classes sont juste des objets qui peuvent créer d'autres objets, et les métaclassees sont juste des classes dont les instances sont des classes. . . . .	9
1-6	Les instances de <i>Workstation</i> ont deux valeurs : leur nom et leur prochain noeud. . . . .	9
1-7	La classe <i>Point</i> en tant qu'objet. . . . .	11
1-8	<i>Class</i> en tant qu'objet. . . . .	11
1-9	A travers le prisme des objets. . . . .	12
1-10	L'envoi d'un message est un processus en deux étapes : la recherche de la méthode et l'exécution. . . . .	12
1-11	La recherche d'une méthode est un processus en deux étapes : il faut d'abord aller dans la classe du receveur puis suivre l'héritage. . . . .	14
1-12	Lorsqu'un message n'est pas trouvé, un autre message est envoyé au receveur pour soutenir l'opération de réflexion. . . . .	14
1-13	Graphique d'héritage complet : Chaque classe hérite finalement de la classe <i>Object</i> . . . . .	17
1-14	Noyau avec lien d'instanciation et d'héritage. . . . .	18
1-15	<i>self</i> représente toujours le receveur. . . . .	19
1-16	Le diagramme séquentiel de <i>self</i> représente toujours le receveur. . . . .	20
1-17	<i>super</i> représente le receveur, mais la recherche de la méthode commence dans la superclasse de la classe de la méthode utilisant <i>super</i> . . . . .	20
1-18	Rôle de la métaclasse pendant la création de l'instance : Application de la résolution de messages simples. . . . .	22
1-19	Rôle de la métaclasse pendant la création de la classe : Application de la résolution du message simple - le lien d'auto-instanciation est suivi. . . . .	23
1-20	Métaclasse abstraite : ses instances (c'est-à-dire la classe <i>Node</i> ) sont abstraites. . . . .	26
1-21	La métaclasse abstraite au travail. . . . .	27
1-22	Une métaclasse <i>WithSingleton</i> : ses instances ne peuvent avoir qu'une seule instance. . . . .	29
1-23	Stockage d'une instance unique. . . . .	29

2-1	Représentation de la structure de la classe. . . . .	38
2-2	Utilisation du décalage pour accéder à l'information. . . . .	38
2-3	Décalage de la variable d'instance demandé à la classe. . . . .	40
2-4	L'offset de la variable d'instance demandé à l'instance elle-même. . . . .	40

Au début des années 90, lorsque j'étais étudiant en master, nous avons appris la méta-programmation objet et nous utilisions Common Lisp Object Systems (CLOS) pour programmer. J'ai adoré ce cours et j'ai réalisé qu'il était assez avancé. À l'époque, j'ai construit un système de démonstration de théorèmes en CLOS. Cependant, au fond de moi, je savais que je ne comprenais pas complètement ce qu'était la classe `Object` ou ce qu'est réellement une méta-classe ; bien sûr, je pouvais répéter le cours et avoir l'air intelligent, mais il y avait cette petite voix qui me disait que je n'étais pas sûr à 100%. Puis par hasard j'ai trouvé l'article de Pierre Cointe et j'ai été bluffé par la simplicité du modèle. J'ai passé 3 jours à réimplémenter le modèle comme un fou parce que c'était trop amusant. Pour moi, c'était la clé de ma compréhension des systèmes réflexifs basés sur les classes. Une fois que j'ai terminé, je suis allée voir mon professeur et lui ai dit qu'elle devait l'enseigner et elle m'a dit de le faire. Depuis lors, je l'enseigne.

Notez que si le projet est historiquement nommé `ObjVLisp`, il n'a rien à voir avec LISP. `ObjVLisp` n'est qu'un petit framework conceptuel mais il fournit une vue condensée et explique les forces en présence dans des systèmes plus grands comme `Pharo` qui partagent le mantra *tout est un objet* que j'aime tant. Ce livre explique la conséquence d'avoir des classes comme objets. En outre, il décrit la conception et les conséquences de l'utilisation d'un noyau minimal réfléchi.

Ce faisant, nous apprendrons profondément sur les objets, l'instanciation de la création d'objets, la recherche de messages, la délégation, l'héritage et bien plus encore.

Je tiens à remercier Christopher Fuhrman pour son important travail de relecture, ainsi que kksbbu et Ren'e-Paul pour leurs suggestions.

— Stéphane Ducasse







# Un noyau minimal réflexif basé sur les classes

*La différence entre les classes et les objets a été soulignée à plusieurs reprises. Dans la vision présentée ici, ces concepts appartiennent à des mondes différents : le texte du programme ne contient que des classes ; à l'exécution, seuls les objets existent. Cette approche n'est pas la seule. L'une des sous-cultures de la programmation orientée objet, influencée par Lisp et illustrée par Smalltalk, considère les classes comme des objets eux-mêmes, qui ont toujours une existence au moment de l'exécution. - B. Meyer, Object-Oriented Software Construction*

Comme l'exprime cette citation, il existe un domaine où les classes sont de véritables objets, des instances d'autres classes. Dans de tels systèmes tels que Smalltalk, Pharo, CLOS, les classes sont décrites par d'autres classes et forment des architectures souvent réflexives, chacune décrivant le niveau précédent. Dans ce chapitre, nous allons explorer un noyau minimal réflexif basé sur les classes, inspiré de ObjVlisp [4]. Dans le chapitre suivant, vous implémenterez pas à pas un tel noyau avec moins de 30 méthodes.

## 1.1 L'inspiration d'ObjVlisp

ObjVlisp a été publié pour la première fois en 1986 alors que les fondements de la programmation orientée objet étaient encore en train d'émerger [4]. ObjVlisp possède des métaclasse explicites et supporte la réutilisation des métaclasse. Il a été inspiré du noyau de Smalltalk-78. Le noyau SOM-DSOM d'IBM est similaire à ObjVlisp tout en étant implémenté en C++ [5]. ObjVlisp est un sous-ensemble du noyau réflexif de CLOS (Common Lisp Object System) puisque CLOS réifie les variables d'instance, les fonctions génériques et la combinaison de méthodes [10][7]. En comparaison avec ObjVlisp, Smalltalk

et Pharo ont des métaclasses implicites et aucune réutilisation de métaclasses sauf par héritage de base [1]. Cependant, ils sont plus stables comme l'expliquent Bouraqadi et al [6][3].

L'étude de ce noyau en vaut vraiment la peine, car il possède les propriétés suivantes :

- Il unifie les classes et les instances (il n'y a qu'une seule structure de données pour représenter tous les objets, classes comprises),
- Il est composé de seulement deux classes `Class` et `Object` (il s'appuie sur des éléments existants tels que les booléens, les tableaux et les chaînes de caractères du langage d'implémentation sous-jacent),
- Elle soulève la question de la régression infinie de la méta-circularité (une classe est une instance d'une autre classe qui est une instance d'une autre classe encore, etc.) et de la manière de la résoudre,
- Elle nécessite de prendre en compte l'allocation, l'initialisation des classes et des objets, le passage des messages ainsi que le processus de bootstrap,
- Il peut être implémenté en moins de 30 méthodes dans Pharo.

N'oubliez pas que ce noyau est auto-décrit. Nous allons commencer à expliquer certains aspects, mais comme tout est lié, vous devrez peut-être lire le chapitre deux fois pour bien comprendre.

## 1.2 Les six postulats d'ObjVLisp

Le noyau original d'ObjVLisp est défini par six postulats [4]. Certains d'entre eux semblent un peu dépassés par les normes modernes, et le 6ème postulat est tout simplement faux comme nous l'expliquerons plus tard (une solution est simple à concevoir et à mettre en œuvre).

Voici les six postulats tels qu'ils sont énoncés dans l'article, par souci de perspective historique.

1. Un objet représente un élément de connaissance et un ensemble de capacités.
2. Le seul protocole pour activer un objet est le passage de messages : un message spécifie la procédure à appliquer (dénotée par son nom, le sélecteur) et ses arguments.
3. Chaque objet appartient à une classe qui spécifie ses données (attributs appelés champs) et son comportement (procédures appelées méthodes). Les objets seront générés dynamiquement à partir de ce modèle; ils sont appelés instances de la classe. Suivant Platon, toutes les instances d'une classe ont la même structure et la même forme, mais diffèrent par les valeurs de leurs variables d'instance communes.

4. Une classe est aussi un objet, instancié par une autre classe, appelée sa métaclasse. En conséquence (P3), à chaque classe est associée une métaclasse qui décrit son comportement en tant qu'objet. La métaclasse primitive initiale est la classe `Class`, construite comme sa propre instance.
5. Une classe peut être définie comme une sous-classe d'une (ou plusieurs) autre(s) classe(s). Ce mécanisme de sous-classement permet le partage des variables d'instance et des méthodes, et s'appelle l'héritage. La classe `Object` représente le comportement le plus commun partagé par tous les objets.
6. Si les variables d'instance possédées par un objet définissent un environnement local, il existe également des variables de classe définissant un environnement global partagé par toutes les instances d'une même classe. Ces variables de classe sont définies au niveau de la métaclasse selon l'équation suivante : variable de classe [un-objet] = variable d'instance [classe d'un-objet].

### 1.3 Aperçu du noyau

Si vous ne saisissez pas entièrement l'aperçu suivant, ne vous inquiétez pas. Ce chapitre complet est là pour s'assurer que vous le comprendrez. Commençons.

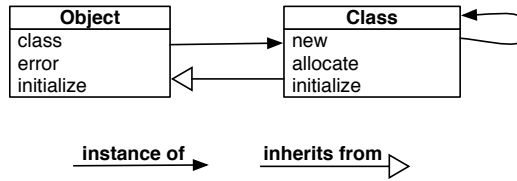
Contrairement à un véritable noyau de langage uniforme, `ObjVlisp` ne considère pas les tableaux, les booléens, les chaînes de caractères, les nombres ou tout autre objet élémentaire comme faisant partie du noyau, comme c'est le cas dans un véritable bootstrap tel que celui de Pharo. Le noyau d'`ObjVlisp` se concentre sur la compréhension des relations fondamentales Classe/Objet.

La figure 1-1 montre les deux classes centrales du noyau :

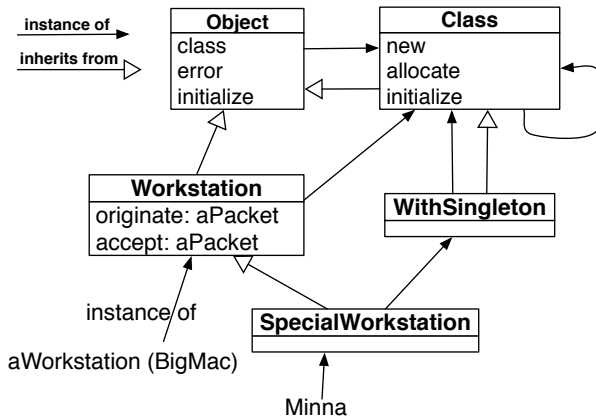
- `Object` qui est la racine du graphe d'héritage et est une instance de `Class`.
- `Class` est la première classe et la racine de l'arbre d'instanciation et l'instance d'elle-même comme nous le verrons plus tard.

La figure 1-2 montre que la classe `Workstation` est une instance de la classe `Class` puisque c'est une classe et qu'elle hérite de `Object` le comportement par défaut que les objets doivent présenter. La classe `WithSingleton` est une instance de la classe `Class` mais en plus elle hérite de `Class`, puisque c'est une métaclasse : ses instances sont des classes. En tant que telle, elle modifie le comportement des classes. La classe `SpecialWorkstation` est une instance de la classe `WithSingleton` et hérite de `Workstation`, puisque ses instances présentent le même comportement que `Workstation`.

Les deux diagrammes 1-1 et 1-2 seront expliqués pas à pas tout au long de ce chapitre.



**Figure 1-1** Le noyau ObjVlisp : un noyau minimal basé sur les classes.

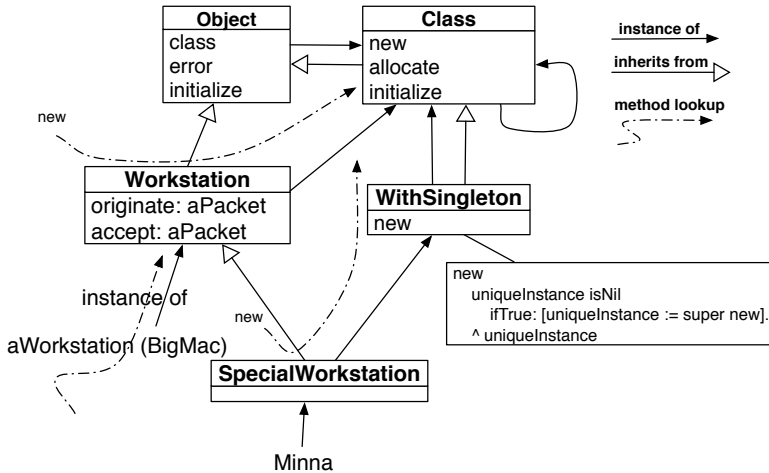


**Figure 1-2** Le noyau avec des métaclasses spécialisées.

**Note** Le point clé de la compréhension d’une telle architecture réflexive est que le passage de messages cherche toujours des méthodes dans la classe du receveur du message et suit ensuite la chaîne d’héritage (Voir Figure 1-3).

La figure 1-3 illustre deux cas principaux :

- Lorsque nous envoyons un message à BigMac ou Minna, la méthode correspondante est recherchée dans leurs classes correspondantes Workstation ou SpecialWorkstation et suit le lien d’héritage jusqu’à Object.
- Lorsque nous envoyons un message aux classes Workstation ou SpecialWorkstation, la méthode correspondante est recherchée dans leur classe, la classe Class et jusqu’à Object.



**Figure 1-3** Comprendre les métaclasse à l'aide du passage de messages.

## 1.4 Instances

Dans ce noyau, il n'existe qu'un seul lien d'instanciation ; il est appliqué à tous les niveaux comme le montre la figure 1-4:

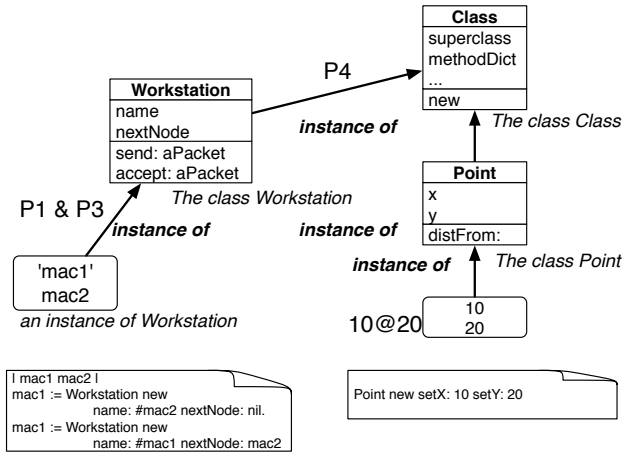
- Les instances de terminaux sont évidemment des objets : une station de travail nommée `mac1` est une instance de la classe `Workstation`, un point `10@20` est une instance de la classe `Point`.
- Les classes sont aussi des objets (instances) d'autres classes : la classe `Workstation` est une instance de la classe `Class`, la classe `Point` est une instance de la classe `Class`.

Dans nos diagrammes, nous représentons les objets (principalement les instances terminales) sous forme de rectangles arrondis avec la liste des valeurs des variables d'instance. Comme les classes sont des objets, *lorsque nous voulons souligner que les classes sont des objets*, nous utilisons la même convention graphique que celle de la figure 1-7.

### Gestion de la récursion infinie

Une classe est un objet. Elle est donc une instance d'une autre classe, sa métaclasse. Cette métaclasse est aussi un objet, une instance d'une métaméta-classe qui est aussi un objet, une instance d'une autre métamétaclasse, etc. Pour arrêter cette potentielle récursion infinie, `ObjVlisp` est similaire aux solutions proposées dans de nombreux systèmes de métaclasse : une instance (par exemple, `Class`) est une instance d'elle-même.

Dans `ObjVlisp` :



**Figure 1-4** Chaîne d'instanciation : les classes sont aussi des objets.

- **Class** est la classe initiale et la métaclasse,
- **Class** est une instance d'elle-même, et, directement ou indirectement, toutes les autres métaclasses sont des instances de **Class**.

Nous verrons plus tard l'implication de cette auto-instantiation au niveau de la structure de la classe elle-même.

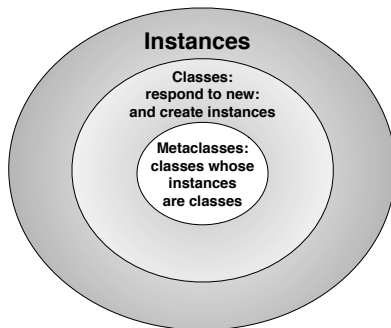
## 1.5 Comprendre les métaclasses

Le modèle unifie les classes et les instances. Il découle des postulats du noyau relatifs aux instances que :

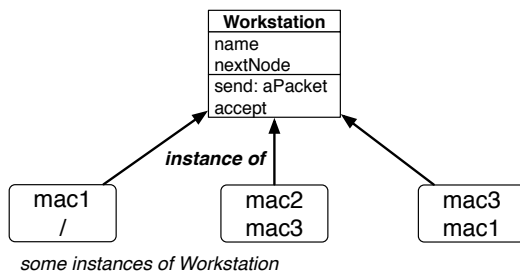
- Tout objet est une instance d'une classe,
- Une classe est un objet instance d'une métaclasse, et
- Une métaclasse est seulement une classe qui génère des classes.

Au niveau de l'implémentation, il n'existe qu'un seul type d'entité : les objets. Il n'y a pas de traitement spécial pour les classes. Les classes sont instanciées selon le même processus que les instances terminales. On leur envoie des messages de la même manière qu'aux autres objets.

Cette unification entre instances et classes ne signifie pas que les objets et les classes ont la même distinction. En effet, tous les objets ne sont pas des classes. En particulier, la seule différence entre une classe et une instance est la capacité de répondre au message de création : **new**. Seule une classe sait y répondre. Ainsi, les métaclasses ne sont que des classes dont les instances sont des classes comme le montre la figure 1-5.



**Figure 1-5** Tout est un objet. Les classes sont juste des objets qui peuvent créer d'autres objets, et les métaclasse sont juste des classes dont les instances sont des classes.



**Figure 1-6** Les instances de `Workstation` ont deux valeurs : leur nom et leur prochain noeud.

## 1.6 Structure de l'instance

Le modèle n'apporte pas vraiment de nouveauté concernant la structure des instances par rapport à des langages tels que Pharo ou Java.

Les variables d'instance sont une séquence ordonnée de variables d'instance définies par une classe. Ces variables d'instance sont partagées par toutes les instances. Les valeurs de ces variables d'instance sont spécifiques à chaque instance. La figure 1-6 montre que les instances de `Workstation` ont deux valeurs : un nom et un nœud suivant.

De plus, nous remarquons qu'un objet possède un pointeur vers sa classe. Comme nous le verrons plus tard lorsque nous aborderons l'héritage, chaque objet possède une variable d'instance `class` (héritée de `Object`) qui pointe vers sa classe.

Notez que cette gestion d'une variable d'instance de classe définie dans `Object` est spécifique au modèle. Dans Pharo par exemple, l'identification de la classe n'est pas gérée comme une variable d'instance déclarée, mais comme

un élément faisant partie de tout objet. Il s'agit d'un index dans une table de classe.

## 1.7 A propos du comportement

Poursuivons avec le comportement de base des instances. Comme dans les langages modernes basés sur les classes, ce noyau doit représenter la manière dont les méthodes sont stockées et recherchées.

Les méthodes appartiennent à une classe. Elles définissent le comportement de toutes les instances de la classe. Elles sont stockées dans un dictionnaire de méthodes qui associe une clé (le sélecteur de méthode) et le corps de la méthode.

Puisque les méthodes sont stockées dans une classe, le dictionnaire de méthodes doit être décrit dans la métaclasse. Ainsi, le dictionnaire de méthodes d'une classe est la *valeur* de la variable d'instance `methodDict` définie sur la métaclasse `Class`. Chaque classe aura son propre dictionnaire de méthodes.

## 1.8 La classe en tant qu'objet

Voici les informations minimales qu'une classe devrait avoir :

- Une liste de variables d'instance pour décrire les valeurs que les instances vont contenir,
- Un dictionnaire de méthodes pour contenir les méthodes,
- Une superclasse pour rechercher les méthodes héritées.

Cet état minimal est similaire à celui de Pharo : la classe `Pharo Behavior` possède un format (description compacte des variables d'instance), un dictionnaire de méthodes et un lien vers une superclasse.

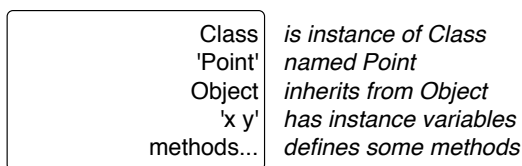
Dans `ObjVLisp`, nous avons un nom pour identifier la classe. En tant que fabriquer d'instance, la métaclasse `Class` possède quatre variables d'instance qui décrivent une classe :

- `name`, le nom de la classe,
- `superclass`, sa superclasse (nous nous limitons à l'héritage simple),
- `i-v`, la liste de ses variables d'instance, et
- `methodDict`, un dictionnaire de méthodes.

Puisqu'une classe est un objet, une classe possède la variable d'instance `class` héritée de `Object` qui fait référence à sa classe comme à n'importe quel objet.

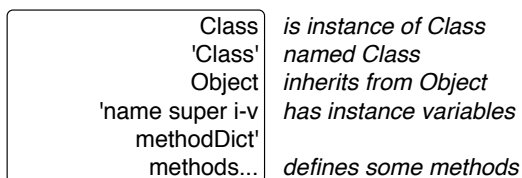


*The class Point*



**Figure 1-7** La classe Point en tant qu'objet.

*The class Class*



**Figure 1-8** Class en tant qu'objet.

### Exemple : classe Point

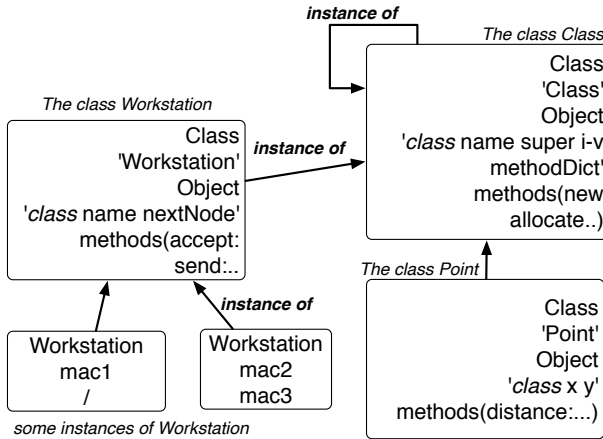
La figure 1-7 montre les valeurs des variables d'instance de la classe Point telle qu'elle est déclarée par le programmeur et avant que l'initialisation de la classe et l'héritage n'aient lieu.

- C'est une instance de la classe Class : il s'agit en effet d'une classe.
- Elle est nommée 'Point'.
- Il hérite de la classe Object.
- Elle possède deux variables d'instance : x et y : après l'héritage, il y aura trois variables d'instance : class, x, et y.
- Il possède un dictionnaire de méthodes.

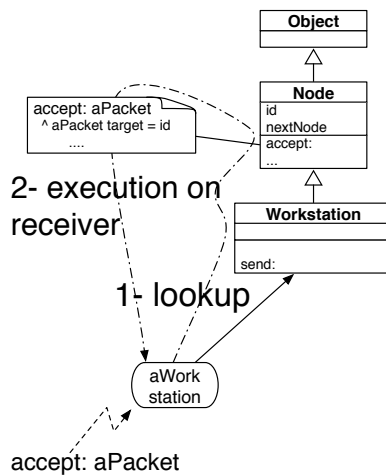
### Exemple : classe Class

La figure 1-8 décrit la classe Class elle-même. En effet, elle est aussi un objet.

- C'est une instance de la classe Class : en effet c'est une classe.
- Elle est nommée 'Class'.
- Elle hérite de la classe Object.
- Elle possède quatre variables d'instance définies localement : name, superclass, i-v, et methodDict.
- Il possède un dictionnaire de méthodes.



**Figure 1-9** A travers le prisme des objets.



**Figure 1-10** L'envoi d'un message est un processus en deux étapes : la recherche de la méthode et l'exécution.

## Tout est un objet

La figure 1-9 décrit une situation typique d'instances terminales, de classes et de métaclasses lorsqu'elle est vue du point de vue des objets. Nous voyons trois niveaux d'instances : les objets terminaux (mac1 et mac2 qui sont des instances de Workstation), les objets de classe (Workstation et Point qui sont des instances de Class) et la métaclasse (Class qui est une instance d'elle-même).

## 1.9 Envoi d'un message

Dans ce noyau, le deuxième postulat stipule que la seule façon d'effectuer des calculs est de passer des messages.

L'envoi d'un message est un processus en deux étapes, comme l'illustre la figure 1-10.

1. Recherche de la méthode : la méthode correspondant au sélecteur est recherchée dans la classe du receveur et ses superclasses.
2. Exécution de la méthode : la méthode est appliquée au receveur. Cela signifie que `self` ou `this` dans la méthode sera lié au receveur.

Conceptuellement, l'envoi d'un message peut être décrit par la composition de fonctions suivante :

```

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

```

### Méthode lookup

Le processus de recherche est maintenant défini conceptuellement comme suit :

1. La recherche commence dans la **classe** du **receveur**.
2. Si la méthode est définie dans cette classe (c'est-à-dire si la méthode est définie dans le dictionnaire des méthodes), elle est retournée.
3. Sinon, la recherche continue dans la superclasse de la classe actuellement explorée.
4. Si aucune méthode n'est trouvée et qu'il n'y a pas de superclasse à explorer (si nous sommes dans la classe `Object`), il s'agit d'une erreur (c'est-à-dire que la méthode n'est pas définie).

La méthode `lookup` parcourt le graphe d'héritage une classe à la fois en utilisant le lien `superclasse`. Voici une description possible de l'algorithme de recherche qui sera utilisé pour les méthodes d'instance et de classe.

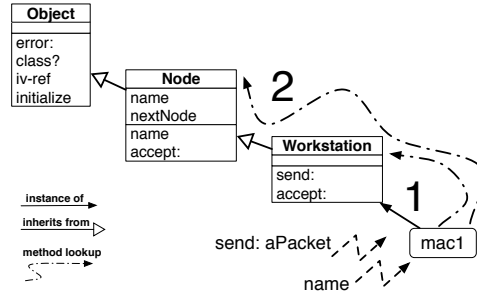
```

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

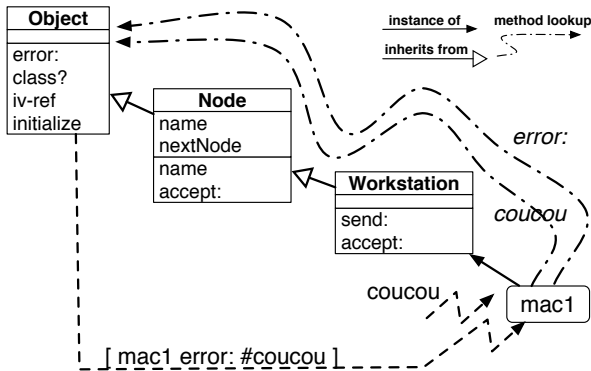
```

### Gestion des erreurs

Lorsque la méthode n'est pas trouvée, le message `error` est envoyé comme indiqué dans la figure 1-12. L'envoi d'un message au lieu de simplement sig-



**Figure 1-11** La recherche d'une méthode est un processus en deux étapes : il faut d'abord aller dans la classe du receveur puis suivre l'héritage.



**Figure 1-12** Lorsqu'un message n'est pas trouvé, un autre message est envoyé au receveur pour soutenir l'opération de réflexion.

naler une erreur à l'aide d'une trace ou d'une exception est une décision de conception essentielle. Dans Pharo, cela se fait par le biais du message `doesNotUnderstand:`, et c'est un crochet de réflexion important. En effet, les classes peuvent définir leur propre implémentation de la méthode `error` et effectuer des actions spécifiques au cas des messages qui ne sont pas compris. Par exemple, il est possible d'implémenter des proxies (objets représentant d'autres objets distants) ou de compiler du code à la volée en redéfinissant localement un tel message.

Il faut maintenant noter que l'algorithme précédent présente une limitation lorsqu'une méthode manquante possède un nombre arbitraire d'arguments. Ils ne sont pas transmis au message `error`. Une meilleure façon de gérer cela est de décomposer l'algorithme différemment comme suit :

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

Et puis nous avons redéfini l'envoi d'un message comme suit :

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

## Remarques

Cette recherche est conceptuellement la même que dans Pharo où toutes les méthodes sont publiques et virtuelles. Il n'y a pas de méthodes liées statiquement ; même les méthodes des classes sont recherchées dynamiquement. Cela permet de définir des mécanismes d'enregistrement très élégants et dynamiques.

Bien que la recherche se fasse au moment de l'exécution, elle est souvent mise en cache. Les langages ont généralement plusieurs systèmes de cache, par exemple, global (classe, sélecteur), un par site d'appel, etc.

## 1.10 Héritage

Dans ce noyau, il y a deux aspects de l'héritage à considérer :

- Un statique pour le cas où les sous-classes obtiennent l'état de la super-classe. Cet héritage de variable d'instance est statique dans le sens où il ne se produit qu'une seule fois au moment de la création de la classe, c'est-à-dire au moment de la compilation.
- Un dynamique pour le comportement où les méthodes sont recherchées pendant l'exécution du programme. Dans ce cas, l'arbre d'héritage est parcouru au moment de l'exécution.

Examinons ces deux aspects.

### Héritage des variables d'instance

L'héritage des variables d'instance se fait au moment de la création de la classe. De ce point de vue, il est statique et réalisé une seule fois. Lorsqu'une classe C est créée, ses variables d'instance sont l'union des variables d'instance de sa superclasse et des variables d'instance définies localement dans

la classe C. Chaque langage définit la sémantique exacte de l'héritage des variables d'instance, par exemple, s'il accepte ou non les variables d'instance portant le même nom. Dans notre modèle, nous décidons d'utiliser la méthode la plus simple : il ne doit pas y avoir de noms en double.

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

Un mot sur l'union : lorsque l'implémentation du langage est basée sur des décalages pour accéder aux variables d'instance, l'union doit s'assurer que l'emplacement des variables d'instance héritées reste ordonné par rapport à la superclasse. En général, nous voulons pouvoir appliquer les méthodes de la superclasse aux sous-classes sans les recopier et les recompiler. En effet, si une méthode utilise une variable à une position donnée dans les listes de variables d'instance, l'application de cette méthode aux instances des sous-classes devrait fonctionner. Dans l'implémentation proposée au chapitre suivant, nous utiliserons des accesseurs et ne supporterons pas l'accès direct aux variables d'instance depuis le corps d'une méthode.

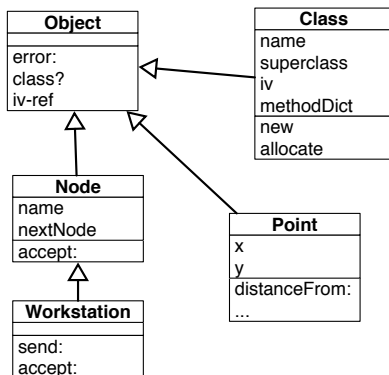
## Recherche de méthode

Comme décrit précédemment dans la section 1.9, les méthodes sont recherchées au moment de l'exécution. Les méthodes définies dans les superclasses sont réutilisées et appliquées aux instances des sous-classes. Contrairement à l'héritage des variables d'instance, cette partie de l'héritage est dynamique, c'est-à-dire qu'elle se produit pendant l'exécution du programme.

### 1.11 Objet : définition du comportement minimal de tout objet

Object représente le comportement minimal que tout objet devrait comprendre, par exemple, renvoyer la classe de l'objet, être capable de gérer les erreurs et initialiser l'objet. C'est pourquoi Object est la racine de la hiérarchie. Selon le langage, Object peut être complexe. Dans notre noyau, elle reste minimale comme nous le montrerons dans le chapitre sur l'implémentation.

La figure 1-13 montre le graphe d'héritage sans la présence d'instanciation. Une station de travail est un objet (c'est-à-dire qu'elle doit au moins comprendre le comportement minimal), donc la classe Workstation hérite directement ou indirectement de la classe Object. Une classe est également un objet (c'est-à-dire qu'elle doit comprendre le comportement minimal), la classe Class hérite donc de la classe Object. En particulier, la variable d'instance `class` (notez la minuscule) est héritée de la classe Object.



**Figure 1-13** Graphique d'héritage complet : Chaque classe hérite finalement de la classe `Object`.

### Remarque.

Dans Pharo, la classe `Object` n'est pas la racine de l'héritage. La racine est en fait `ProtoObject`, et `Object` en hérite. La plupart des classes héritent toujours de `Object`. L'objectif de conception de `ProtoObject` est particulier : générer le plus d'erreurs possible. Ces erreurs peuvent ensuite être capturées via la redéfinition de `doesNotUnderstand:` et peuvent supporter différents scénarios tels que l'implémentation d'un proxy.

## 1.12 Héritage et instanciation ensemble

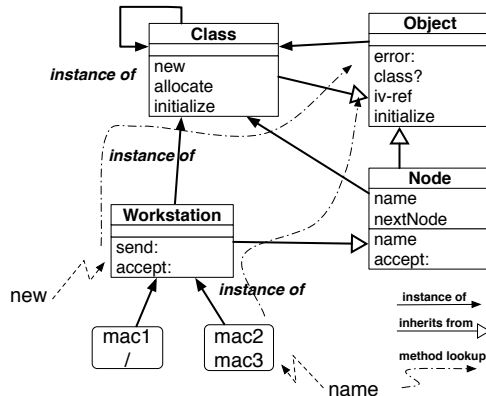
Maintenant que nous avons vu les graphes d'instanciation et d'héritage, nous pouvons examiner l'image complète. La figure 1-14 montre les graphes et en particulier comment ces graphes sont utilisés pendant la résolution des messages :

- le lien d'instanciation est utilisé pour trouver la classe de départ afin de rechercher toutes les méthodes associées au message reçu.
- le lien d'héritage est utilisé pour trouver les méthodes héritées.

Ce processus est le même lorsque nous envoyons des messages aux classes elles-mêmes. Il n'y a pas de différence entre envoyer un message à un objet ou à une classe. Le système effectue *toujours* les mêmes étapes.

## 1.13 Révision de la sémantique du self et du super

Comme notre expérience nous a montré que même certains auteurs de livres se trompent sur la sémantique essentielle de la programmation orientée objet, nous passons en revue ici quelques faits que les programmeurs familiers



**Figure 1-14** Noyau avec lien d'instanciation et d'héritage.

de la programmation orientée objet devraient maîtriser. Pour plus d'informations, reportez-vous à *Pharo Par l'Exemple* ou au Pharo Mooc disponible sur <http://mooc.pharo.org>.

Comme expliqué dans la section 1.9, l'envoi d'un message à un objet déclenche toujours la recherche de la méthode correspondante dans la classe du receveur.

Il faut maintenant distinguer deux cas : **self** et **super**. Dans le corps d'une méthode, tant **self** (également appelé **this** dans des langages comme Java) que **super** représentent toujours le receveur du message. Oui, vous avez bien lu, les deux **self** et **super** représentent toujours le receveur !

La différence réside dans la classe à partir de laquelle la recherche commence :

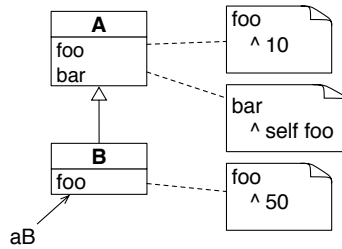
- Pour **self**. Lorsqu'un message est envoyé à **self**, la recherche de la méthode à exécuter commence dans la classe du receveur.

Lorsqu'un message est envoyé à **super**, la recherche commence dans la superclasse de la classe de la méthode.

- Pour **super**. Lorsqu'un message est envoyé à **super**, la recherche de la méthode commence dans la superclasse de la classe contenant l'expression **super**.

Cette distinction entre **self** et **super** est nécessaire pour gérer le cas où une méthode est redéfinie localement dans une classe mais que vous devez invoquer le comportement défini dans ses superclasses. Notez que la méthode de la superclasse peut être définie non pas dans une superclasse directe mais dans une classe ancêtre, il est donc nécessaire d'effectuer une recherche de méthode et cette recherche de méthode doit commencer au-dessus de





**Figure 1-15** self représente toujours le receveur.

la méthode redéfinie (ici la méthode contenant l'expression **super**). D'où le nom **super**.

Notez que la recherche d'une méthode dans le cas de **super** ne cherche pas dans la superclasse de la classe du receveur, car cela signifierait qu'elle pourrait tourner en boucle à l'infini dans le cas d'un arbre d'héritage avec trois classes.

En regardant la figure 1-15, nous voyons que le point clé est que `B new bar` renvoie 50 puisque la méthode est recherchée dynamiquement et **self** représente le receveur, c'est-à-dire l'instance de la classe B. Ce qu'il est important de voir, c'est que les envois de **self** agissent comme un crochet et que le code des sous-classes peut être injecté dans le code de la superclasse.

```

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

```

Pour **super**, la situation décrite dans la figure 1-17 montre que **super** représente le receveur, mais que lorsque **super** est le receveur d'un message, la méthode est recherchée différemment (en partant de la superclasse de la classe utilisant **super**) ; ainsi `R new bar` renvoie 100, mais ni 20 ni 60.

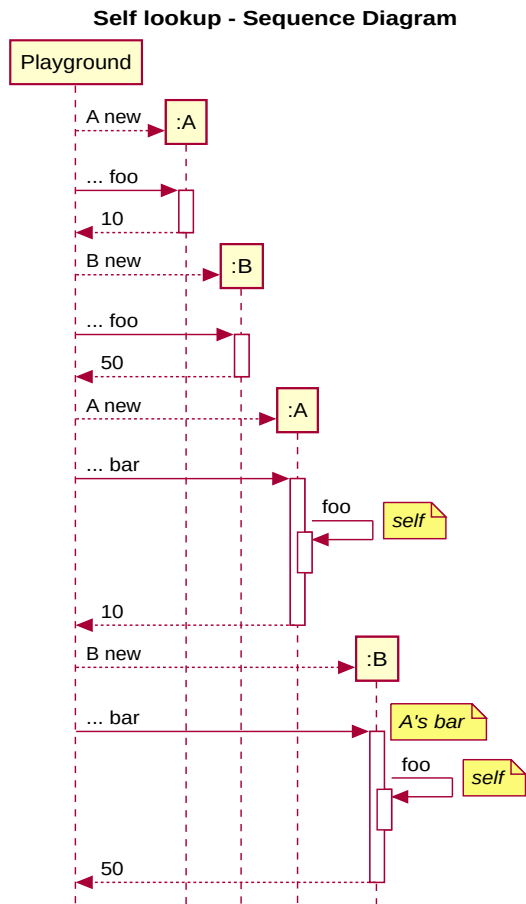
```

Q new bar
>>> 20
R new bar
>>> 100

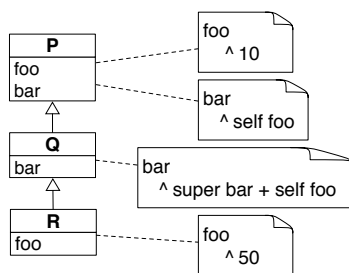
```

En conclusion, nous pouvons dire que **self** est dynamique et **super** statique. Expliquons ce point de vue :

- Lorsqu'on envoie un message à **self**, la recherche de la méthode commence dans la classe du receveur. **self** est lié au moment de l'exécu-



**Figure 1-16** Le diagramme séquentiel de self représente toujours le receveur.



**Figure 1-17** super représente le receveur, mais la recherche de la méthode commence dans la superclasse de la classe de la méthode utilisant super.

tion. Nous ne connaissons pas sa valeur avant le moment de l'exécution.

- `super` est statique dans le sens où, alors que l'objet vers lequel il pointera n'est connu qu'au moment de l'exécution, l'endroit où chercher la méthode est connu au moment de la compilation : il doit commencer à chercher dans la classe au-dessus de celle qui contient `super`.

## 1.14 Création d'objets

Nous sommes maintenant prêts à comprendre la création d'objets. Dans ce modèle, il n'y a qu'une seule façon de créer des instances : nous devons envoyer le message `new` à la classe avec une spécification des valeurs des variables d'instance comme argument.

## 1.15 Création d'instances de la classe `Point`

Les exemples suivants montrent plusieurs instanciations de points. Ce que nous voyons, c'est que le modèle hérite de la tradition Lisp de passer des arguments en utilisant des clés et des valeurs, et que l'ordre des arguments n'est pas important.

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

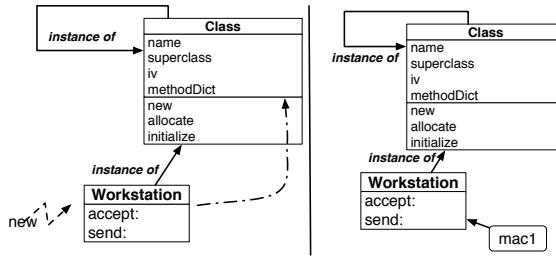
Lorsqu'aucune valeur n'est spécifiée, la valeur d'une variable d'instance est initialisée à `nil`. CLOS fournit la notion de valeurs par défaut pour l'initialisation des variables d'instance. Elle peut être ajoutée à `ObjVlisp` comme un exercice et n'apporte pas de difficultés conceptuelles.

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

Lorsque le même argument est passé plusieurs fois, alors l'implémentation prend la première occurrence.

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

Nous ne devrions pas trop nous inquiéter de ces détails : Le point est que nous pouvons passer plusieurs arguments avec une balise pour les identifier.



**Figure 1-18** Rôle de la métaclasse pendant la création de l'instance : Application de la résolution de messages simples.

## 1.16 Création de la classe Point instance de Class

Puisque la classe Point est une instance de la classe Class, pour la créer, nous devons envoyer le message new à la classe comme suit :

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

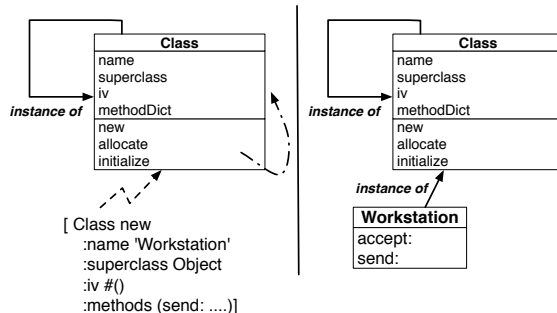
Ce qui est intéressant à voir ici, c'est que nous utilisons exactement la même méthode pour créer une instance de la classe Point que la classe elle-même. Notez que la possibilité d'avoir la même façon de créer des objets ou des classes est également due au fait que les arguments sont spécifiés en utilisant une liste de paires.

Une implémentation pourrait avoir deux messages différents pour créer des instances et des classes. Dès lors que les mêmes méthodes new, allocate, ou initialize sont impliquées, l'essence de la création d'objet est similaire et uniforme.

### Création d'instances : Rôle de la métaclasse

Le diagramme suivant (Figure 1-18) montre que malgré ce que l'on pourrait croire, lorsque nous créons une instance d'un terminal, la métaclasse Class est impliquée dans le processus. En effet, nous envoyons le message new à la classe, pour résoudre ce message, le système va chercher la méthode dans la classe du receveur (ici Workstation) qui est la métaclasse Class. La méthode new est trouvée dans la métaclasse et appliquée au receveur, la classe Workstation. Son effet est de créer une instance de la classe Workstation.

La même chose se produit lors de la création d'une classe. La figure 1-19 montre le processus. Nous envoyons un message, cette fois à la classe Class. Le système ne fait pas d'exception et pour résoudre le message, il cherche



**Figure 1-19** Rôle de la métaclasse pendant la création de la classe : Application de la résolution du message simple - le lien d'auto-instanciation est suivi.

la méthode dans la classe du receveur. La classe du receveur est elle-même, donc la méthode `new` trouvée dans `Class` est appliquée à `Class` (le receveur du message), et une nouvelle classe est créée.

## **new = allouer et initialiser**

La création d'une instance est la composition de deux actions : un message d'allocation de mémoire `allocate` et un message d'initialisation d'objet `initialize`.

En syntaxe Pharo, cela signifie :

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

Nous devrions voir ce qui suit :

- Le message `new` est un message envoyé à une classe. La méthode `new` est une méthode de classe.
- Le message `allocate` est un message envoyé à une classe. La méthode `allocate` est une méthode de classe.
- Le message `initialize:` sera exécuté sur toute instance nouvellement créée. S'il est envoyé à une classe, une méthode de classe `initialize:` sera impliquée. S'il est envoyé à un objet d'un terminal, une méthode d'instance `initialize:` sera exécutée (définie dans `Object`).

## **Allocation d'objet : le message allocate**

Allouer un objet signifie allouer suffisamment d'espace à l'état de l'objet mais il y a plus : les instances doivent être marquées avec leur nom de classe ou leur id. Il existe un invariant dans ce modèle et en général dans les modèles de programmation orientés objet. Chaque objet doit avoir un identifiant

pour sa classe, sinon le système s'arrêtera lorsqu'il essaiera de résoudre un message.

L'allocation d'objet doit retourner une instance nouvellement créée avec :

- des variables d'instance vides (pointant vers nil par exemple) ;
- un identifiant de sa classe.

Dans notre modèle, le marquage d'un objet comme instance d'une classe est effectué en fixant la valeur de la variable d'instance `class` héritée de `Object`. Dans Pharo, cette information n'est pas enregistrée comme une variable d'instance mais encodée dans la représentation interne de l'objet dans la machine virtuelle.

La méthode `allocate` est définie sur la métaclasse `Class`. Voici quelques exemples d'allocation.

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

Une allocation de point alloue trois slots : un pour la classe et deux pour les valeurs `x` et `y`.

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

L'allocation pour un objet représentant une classe alloue six slots : un pour la classe et un pour chacune des variables d'instance de la classe : `name`, `super`, `iv`, `keywords`, et `methodDict`.

## Initialisation d'objet

L'initialisation d'un objet est le processus consistant à passer des arguments sous forme de paires clé/valeur et à affecter la ou les valeurs à la ou les variables d'instance correspondantes.

Ce processus est illustré dans l'extrait suivant. Une instance de la classe `Point` est créée et les paires clé/valeur `(:y 6)` et `(:x 24)` sont spécifiées. L'instance est créée et elle reçoit le message `initialize:` avec les couples clé/valeur. La méthode `initialize:` est chargée de définir les variables correspondantes dans le receveur.

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

Lorsqu'un objet est initialisé en tant qu'instance d'un terminal, deux actions sont effectuées :

- Premièrement, nous devons obtenir les valeurs spécifiées lors de la création, c'est-à-dire obtenir que la valeur `y` est 6 et la valeur `x` est 24,

- Ensuite, nous devons affecter ces valeurs aux variables d'instance correspondantes de l'objet créé.

## Initialisation de la classe

Pendant son initialisation, une classe doit effectuer plusieurs étapes :

- Tout d'abord, comme pour toute initialisation, elle doit obtenir les arguments et les affecter à leurs variables d'instance correspondantes. Ceci est essentiellement mis en œuvre en invoquant la méthode `initialize` de `Object` via un appel `super`, puisque `Object` est la super-classe de `Class`.
- Ensuite, l'héritage des variables d'instance doit être effectué. Avant cette étape, la variable d'instance de la classe `iv` ne contient que les variables d'instance qui sont définies localement. Après cette étape, la variable d'instance `iv` contiendra toutes les variables d'instance héritées et locales. En particulier, c'est ici que la variable d'instance `class` héritée de `Object` est ajoutée à la liste des variables d'instance de la sous-classe de `Object`.
- Troisièmement, la classe doit être déclarée comme un pool de classes ou un espace de noms afin que les programmeurs puissent y accéder via son nom.

## 1.17 La classe Class

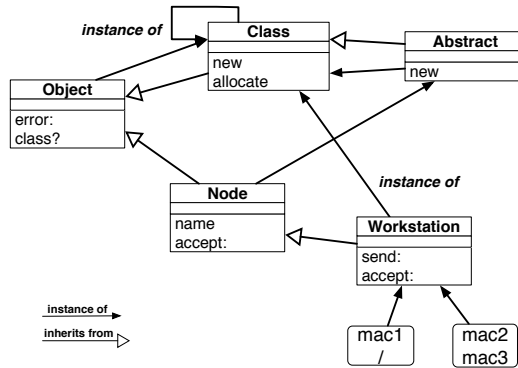
Nous comprenons maintenant mieux ce qu'est la classe `Class` :

- C'est la métaclasse initiale et la classe initiale.
- Elle définit le comportement de toutes les métaclasses.
- Elle définit le comportement de toutes les classes.

En particulier, les métaclasses définissent trois messages liés à la création d'instances.

- Le message `new`, qui crée une instance initialisée de la classe. Il alloue l'instance en utilisant le message de classe `allocate` puis l'initialise en envoyant le message `initialize` : à cette instance.
- Le message `allocate`. Comme le message `new`, il s'agit d'un message de classe. Il alloue la structure de l'objet nouvellement créé.
- Enfin le message `initialize` :. Ce message a deux définitions, une sur `Object` et une sur `Class`.

Il y a une différence entre la méthode `initialize` : exécutée lors de toute création d'instance et la méthode `initialize` : exécutée uniquement lorsque l'instance créée est une classe.



**Figure 1-20** Métaclasse abstraite : ses instances (c'est-à-dire la classe Node) sont abstraites.

- La première est une méthode définie sur la classe de l'objet et potentiellement héritée de Object. Cette méthode `initialize:` extrait simplement les valeurs correspondant à chaque variable d'instance de la liste des arguments et les place dans les variables d'instance correspondantes.
- La méthode `initialize:` de la classe est exécutée lorsqu'une nouvelle instance représentant une classe est exécutée. Le message `initialize:` est envoyé à l'objet nouvellement créé mais sa spécialisation pour les classes sera trouvée lors de la recherche de méthode et elle sera exécutée. Habituellement, cette méthode invoque les méthodes par défaut, car les paramètres de classe doivent être extraits de la liste d'arguments et placés dans leurs variables d'instance correspondantes. Mais en plus, l'héritage des variables d'instance et la déclaration de la classe dans l'espace de noms de la classe sont effectués.

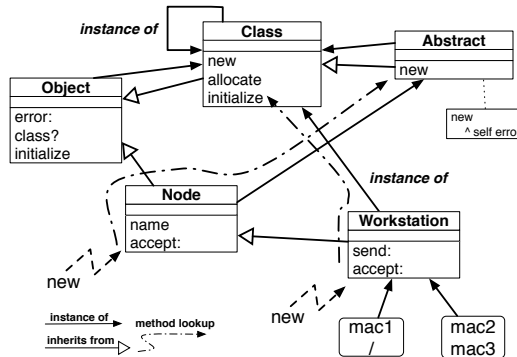
## 1.18 Définition d'une nouvelle métaclasse

Nous pouvons maintenant étudier comment nous pouvons ajouter de nouvelles métaclasses et voir comment le système les traite. Créer une nouvelle métaclasse est simple ; il suffit d'hériter d'une métaclasse existante. Peut-être que cela est évident pour vous, mais c'est ce que nous allons vérifier maintenant.

### Abstrait

Imaginons que nous voulions définir des classes abstraites. Nous déclarons qu'une classe est abstraite si elle ne peut pas créer d'instances. Pour contrôler la création d'instances d'une classe, nous devons définir une nouvelle





**Figure 1-21** La métaclass abstraite au travail.

métaclass qui l'interdit. Nous allons donc définir une métaclass dont les instances (classes abstraites) ne peuvent pas créer d'instances.

Nous créons une nouvelle métaclass nommée `AbstractMetaclass` qui hérite de `Class` et nous redéfinissons la méthode `new` dans cette métaclass pour qu'elle lève une erreur (comme le montre la figure 1-20). L'extrait de code suivant définit cette nouvelle métaclass.

```
[ Class new
  :name 'AbstractMetaclass'
  :super 'Class'

AbstractMetaclass
  addMethod: #new
  body: [ :receiver :initargs | receiver error: 'Cannot create
    instance of class' ]
```

Deux faits décrivent les relations entre cette métaclass et la classe `Class` :

- `AbstractMetaclass` est une classe, une instance de `Class`.
- `AbstractMetaclass` définit le comportement de la classe : Elle hérite de la `Class`.

Nous pouvons maintenant définir une classe abstraite `Node`.

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

L'envoi d'un message `new` à la classe `Node` soulèvera une erreur.

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

Une sous-classe de `Node`, par exemple `Workstation`, peut être une classe concrète en étant une instance de `Class` au lieu de `AbstractMetaclass` mais en héritant toujours de `Node`. Ce que nous voyons dans la figure 1-21 est qu'il

y a deux liens, l'instanciation et l'héritage. La recherche de méthode les suit comme nous l'avons présenté précédemment. Elle commence toujours dans la classe du receveur et suit le chemin de l'héritage.

Ce qu'il faut comprendre, c'est que lorsque nous envoyons le message `new` à la classe `Workstation`, nous cherchons d'abord les méthodes dans la méta-classe `Class`. Lorsque nous envoyons le message `new` à la classe `Node`, nous cherchons dans sa classe : `AbstractMetaClass` comme le montre la figure 1-21. En fait, nous faisons ce que nous faisons pour toute instance : nous regardons dans la classe du receveur.

Une méthode de classe est juste implémentée et suit la même sémantique que les méthodes d'instance : L'envoi du message `error` à la classe `Node` commence dans `AbstractMetaClass`. Comme nous ne l'avons pas redéfini localement et qu'il ne s'y trouve pas, la recherche se poursuit dans la super-classe de `AbstractClass` : la classe `Class` puis la superclasse de la classe `Class`, la classe `Object`.

## 1.19 À propos de l'état des classes

Imaginons que nous définissions une métaclasse `WithSingleton` dont les instances sont des classes qui auront une instance unique. La situation est décrite dans la figure 1-22. La classe `WithSingleton` hérite de `Class` puisqu'elle veut réutiliser tous les mécanismes de la classe. Elle est également une instance de la classe `Class`, puisque `WithSingleton` est une classe et qu'elle peut créer des instances. La classe `Node` est une instance de la classe `WithSingleton`. Lorsqu'elle reçoit le message `new`, la méthode `new` définie dans la classe `WithSingleton` est exécutée. Si la variable d'instance unique est nulle, il invoque le comportement défini dans la classe `Class`, le stocke dans la variable d'instance unique et le renvoie.

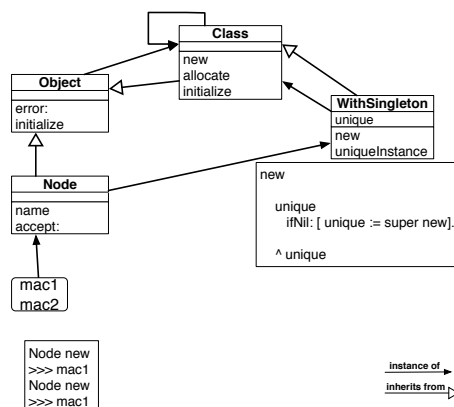
Il reste plusieurs questions à poser et auxquelles il faut répondre.

- Quelle est la liste des variables d'instance pour la métaclasse `WithSingleton` ?
- Où est stockée l'instance unique de la classe `Node` ou `Process` ?

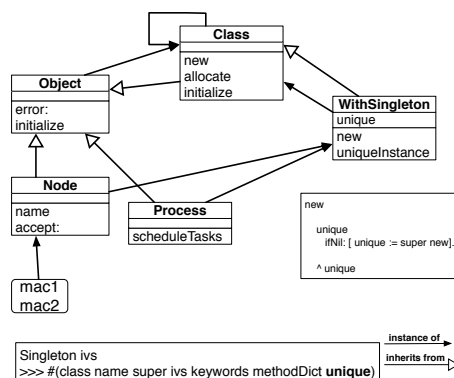
### Variable d'instance de `WithSingleton`

Comme pour toute classe, une sous-classe obtient ses variables d'instance ainsi que les variables d'instance de sa super-classe. Par conséquent, les variables d'instance de `WithSingleton` sont les mêmes que celles de `Class`, et elle possède également unique 1-23.

```
[ Singleton objIVs
>>> #(class name super ivs keywords methodDict unique)
```



**Figure 1-22** Une métaclasse WithSingleton : ses instances ne peuvent avoir qu'une seule instance.



**Figure 1-23** Stockage d'une instance unique.

## Où est stocké le singleton ?

Chaque instance de classe de WithSingleton aura une valeur supplémentaire après son dictionnaire de méthodes. C'est là que le singleton réel de la classe est stocké. Les classes Node et Processor sont des instances de la métaclasse WithSingleton. Elles ont donc un champ supplémentaire dans leur structure pour contenir les valeurs uniques des variables d'instance. Chaque instance de WithSingleton aura sa propre valeur : l'instance de la classe jouant le rôle de singleton.

## 1.20 À propos du 6e postulat

Comme mentionné au début de ce chapitre, le 6ème postulat d'ObjVLisp est erroné. Relisons-le : *Si les variables d'instance possédées par un objet définissent un environnement local, il existe également des variables de classe définissant un environnement global partagé par toutes les instances d'une même classe. Ces variables de classe sont définies au niveau de la métaclasse selon l'équation suivante : variable de classe [un-objet] = variable d'instance [classe d'un-objet].*

Cela dit que les variables d'instance de classe sont équivalentes à des variables partagées entre les instances, et c'est faux. Voyons cela. Selon le 6ème postulat, une variable partagée entre instances est égale à une variable d'instance de la classe. La définition n'est pas totalement claire, regardons donc un exemple donné dans l'article.

### Illustration du problème

Imaginons que l'on souhaite que le caractère constant '\*' soit une variable de classe partagée par tous les points d'une même classe. Nous redéfinissons la classe Point comme précédemment, mais dont la métaclasse (appelons-la MetaPoint) spécifie ce caractère commun. Par exemple, si un point a une variable partagée nommée char, cette variable d'instance doit être définie dans la classe de la classe Point appelée MetaPoint. L'auteur propose de définir une nouvelle métaclasse MetaPoint pour contenir une nouvelle variable d'instance pour représenter une variable partagée entre les points.

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

Il propose ensuite de l'utiliser comme suit :

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

La classe Point peut définir une méthode qui accède au caractère en passant simplement au niveau de la classe. Pourquoi cette approche est-elle mauvaise ? Parce qu'elle mélange les niveaux. La variable d'instance char n'est pas une information de classe. Elle décrit les instances terminal et non l'instance de la métaclasse. Pourquoi la *métaclasse* MetaPoint aurait-elle besoin d'une variable d'instance char ?

## La solution

La solution consiste à conserver la variable partagée `char` dans une liste de variables partagées de la classe `Point`. Toute instance de point peut accéder à cette variable. L'implication est qu'une *classe* devrait avoir des informations supplémentaires pour la décrire. C'est-à-dire une variable d'instance `sharedVariable` contenant des paires, c'est-à-dire la variable et sa valeur. Nous devrions alors être en mesure d'écrire :

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

Par conséquent, la métaclasse `Class` doit obtenir une variable d'instance supplémentaire nommée `sharedivs`, et chacune de ses instances (les classes `Point`, `Node`, `Object`) peut avoir différentes *valeurs*. Ces valeurs peuvent être partagées entre leurs instances par le compilateur.

Ce que nous voyons, c'est que `sharedivs` fait partie du vocabulaire `Class` et que nous n'avons pas besoin d'une métaclasse supplémentaire chaque fois que nous voulons partager une variable. Cette conception est similaire à celle de `Pharo` où une classe possède une variable d'instance `classVariable` contenant des variables partagées dans toutes les sous-classes de la classe qui la définit.

## 1.21 Conclusion

Nous avons présenté un petit noyau composé de deux classes : `Object`, la racine de l'arbre d'héritage et `Class`, la première métaclasse racine de l'arbre d'instanciation. Nous avons revu tous les points clés liés à la recherche de méthodes, à la création et à l'initialisation des objets et des classes. Dans le chapitre suivant, nous vous proposons d'implémenter un tel noyau.

## Lectures complémentaires

Le noyau présenté dans ce chapitre est un noyau avec des métaclasses explicites et en tant que tel, il n'est pas une panacée. En effet, il entraîne des problèmes de composition de métaclasses comme expliqué dans l'excellent article de Bouraqadi et al. [3] ou [5].



# Construction d'un noyau minimal réflexif basé sur des classes

L'objectif de ce chapitre est de vous aider à implémenter pas à pas le modèle ObjVlisp expliqué dans le chapitre précédent. ObjVlisp a été conçu par Pierre Cointe, qui s'est inspiré du noyau de Smalltalk-78. Il possède des métaclasse explicites et il est composé de deux classes `Object` et `Class`.

## 2.1 Objectifs

Au cours du chapitre précédent, vous avez vu les points principaux du modèle ObjVlisp, maintenant vous allez l'implémenter. Les objectifs de cette implémentation sont de donner une compréhension concrète des concepts présentés précédemment. Voici quelques points que vous pourrez approfondir en écrivant l'implémentation :

- Quelle est la structure possible d'un objet ?
- Qu'est-ce que l'allocation et l'initialisation des objets ?
- Qu'est-ce que l'initialisation des classes ?
- Quelle est la sémantique de la méthode lookup ?
- Qu'est-ce qu'un noyau réflexif ?
- Quels sont les rôles des classes `Class` et `Object` ?
- Quel est le rôle d'une métaclasse ?

## 2.2 Préparation

Dans cette section, nous discutons de la configuration que vous allez utiliser, des choix d'implémentation et des conventions que nous allons suivre tout au long de ce chapitre.

### La mise en place de Pharo

Vous devez télécharger et installer Pharo à partir de <https://pharo.org/>. Vous avez besoin d'une machine virtuelle, de l'image du couple et des modifications. Vous pouvez utiliser <https://get.pharo.org> pour obtenir un script de téléchargement de Pharo.

La version actuelle que vous pouvez utiliser est Pharo 7.0.

```
[ wget -O- get.pharo.org/70+vm | bash
```

Vous pouvez utiliser le livre Pharo Par l'Exemple sur <https://books.pharo.org/pharo-by-example/> pour une vue d'ensemble de la syntaxe et du système.

### Obtenir les définitions de l'infrastructure

Toutes les définitions nécessaires sont fournies sous la forme d'un paquet Monticello. Il contient toutes les classes, les catégories de méthodes et les signatures des méthodes que vous devez implémenter. Il fournit des fonctionnalités supplémentaires telles qu'un inspecteur dédié et quelques méthodes supplémentaires qui vous faciliteront la vie et vous aideront à vous concentrer sur l'essence du modèle. Il contient également tous les tests des fonctionnalités que vous devez implémenter.

Pour charger le code, exécutez l'expression suivante :

```
[ Metacello new
  baseline: 'ObjV';
  repository: 'github://Ducasse/ObjVLispSkeleton/tree/master/src';
  load
```

### Exécution des tests

Pour chaque fonctionnalité, vous devrez exécuter des tests.

Par exemple, pour exécuter un test particulier nommé `testPrimitiveStructure`,

- évaluez l'expression `(ObjTest selector : #testPrimitiveStructure) run`, ou bien
- cliquez sur l'icône de la méthode nommée `testPrimitiveStructure`.



Notez que puisque vous êtes en train de développer le noyau, pour le tester nous avons implémenté manuellement quelques mocks des classes et du noyau. C'est la méthode de configuration des classes de test qui construisent ce faux noyau. Maintenant, faites attention car les configurations prennent souvent des raccourcis, donc ne les copiez pas aveuglément.

## 2.3 Conventions de nommage

Nous utilisons les conventions suivantes : nous nommons *primitives* toutes les méthodes Pharo qui participent à la construction d'ObjVLisp. Ces primitives sont principalement implémentées comme des méthodes de la classe Obj. Notez que dans une implémentation Lisp de telles primitives seraient juste des expressions lambda, dans une implémentation C de telles primitives seraient représentées par des fonctions C.

Pour vous aider à faire la distinction entre les classes dans le langage d'implémentation (Pharo) et le modèle ObjVLisp, nous **préfixons toutes les classes** ObjVLisp par **Obj**. Enfin, certaines des **primitives** cruciales et déroutantes (principalement celles de la structure de classe) sont toutes préfixées par **obj**. Par exemple, la primitive qui renvoie l'identifiant de classe d'une **objInstance** est nommée **objClassId**. Nous parlons également d'**objInstances**, d'**objObjects** et d'**objClasses** pour faire référence aux instances, objets ou classes spécifiques définis dans ObjVLisp.

## 2.4 Hériter de la classe Array

Nous ne voulons pas implémenter un scanner, un parseur et un compilateur pour ObjVLisp mais nous concentrer sur l'essence du langage. C'est pourquoi nous avons choisi d'utiliser autant que possible le langage d'implémentation, ici Pharo. Comme Pharo ne supporte pas la définition de macro, nous utiliserons autant que possible les classes existantes pour éviter des problèmes syntaxiques supplémentaires.

Dans notre implémentation, chaque objet dans le monde ObjVLisp est une instance de la classe Obj. La classe Obj est une sous-classe de Array.

Puisque Obj est une sous-classe de Array, `10 15` est une `objInstance` de la classe `ObjPoint` qui est aussi une instance de `array` de la classe `Pharo ObjClass`.

Comme nous allons le voir :

- `10 15` représente un `objPoint (10,15)`. C'est une `objInstance` de la classe `ObjPoint`.
- `10 15` est le tableau qui représente l'`objclass ObjPoint`.

## À propos des choix de représentation

Vous pouvez sauter cette discussion lors d'une première lecture. Nous aurions pu implémenter la fonctionnalité ObjVLisp au niveau d'une classe nommée `Obj` héritant simplement de `Object`. Cependant, pour utiliser la primitive ObjVLisp (une méthode Pharo) `objInstanceVariableValue : anObject for : anInstanceVariable` qui renvoie la valeur de la variable d'instance dans `anObject`, nous aurions été obligés d'écrire l'expression suivante :

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

Nous avons choisi de représenter tout objet ObjVLisp par un tableau et de définir la fonctionnalité ObjVLisp dans le côté instance de la classe `Obj` (une sous-classe de `Array`). De cette façon, nous pouvons écrire de manière plus naturelle et plus lisible la fonctionnalité précédente comme suit :

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

## 2.5 Faciliter l'accès à la classe objclass

Nous avons besoin d'un moyen de stocker et d'accéder aux classes ObjVLisp. Comme solution, au niveau de la classe Pharo `Obj` nous avons défini un dictionnaire contenant les classes définies. Ce dictionnaire fait office d'espace de noms pour notre langage. Nous avons défini les méthodes suivantes pour stocker et accéder aux classes définies.

- `declareClass: anObjClass` stocke l'objInstance `anObjClass` donnée en argument dans le référentiel de classes (ici un dictionnaire dont les clés sont les noms de classes et les valeurs les classes ObjVLisp elles-mêmes).
- `giveClassNameed: aSymbol` renvoie la classe ObjVLisp nommée `aSymbol` si elle existe. La classe doit avoir été déclarée auparavant.

Avec de telles méthodes, nous pouvons écrire du code comme le suivant qui recherche la classe de la classe `ObjPoint`.

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

Pour rendre l'accès aux classes moins lourd, nous avons également implémenté un raccourci : Nous piègeons les messages non compris envoyés à `Obj` et consultons le dictionnaire des classes définies. Puisque `ObjPoint` est un message inconnu, ce même code est alors écrit comme :

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

Maintenant vous êtes prêt à commencer.

## 2.6 Structure et primitives

La première question est de savoir comment représenter les objets. Nous devons nous mettre d'accord sur une représentation initiale. Dans cette implémentation, nous avons choisi de représenter les objInstances comme des tableaux (instances de Obj une sous-classe de Array). Dans ce qui suit, nous utilisons le terme tableau pour parler des instances de la classe Obj.

### Votre travail.

Vérifiez que la classe Obj existe et hérite de Array.

## 2.7 Structure d'une classe

Le premier objet que nous allons créer est la classe ObjClass. Par conséquent, nous nous concentrons maintenant sur la structure minimale des classes de notre langage.

Une objInstance représentant une classe a la structure suivante : un identifiant de sa classe, un nom, un identifiant de sa superclasse (nous limitons le modèle à l'héritage simple), une liste de variables d'instance, une liste de mots-clés d'initialisation, et un dictionnaire de méthodes.

Par exemple la classe ObjPoint a donc la structure suivante :

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

Cela signifie que ObjPoint est une instance de ObjClass, est nommé #ObjPoint, hérite d'une classe nommée ObjObject, possède trois variables d'instance, deux mots-clés d'initialisation et un dictionnaire de méthodes non initialisées. Pour accéder à cette structure, nous définissons quelques primitives comme le montre la figure 2-1. La figure 2-2 montre comment les off-sets sont utilisés pour accéder de manière contrôlée aux informations brutes de ObjClass.

### Votre travail.

Les méthodes de test de la classe RawObjTest qui sont dans les catégories 'step1-tests-structure of objects' et 'step2-tests-structure of classes' donnent quelques exemples d'accès à la structure.

```
[ RawObjTest >> testPrimitiveStructureObjClassId
  "(self selector: #testPrimitiveStructureObjClassId) run"

  self assert: (pointClass objClassId = #ObjClass).
```

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

**Figure 2-1** Représentation de la structure de la classe.

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

name    superclass    instancevariables

objName  
^ self at: self offsetForName

**Figure 2-2** Utilisation du décalage pour accéder à l'information.

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

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

Implémentez les primitives manquantes pour exécuter les tests suivants  
testPrimitiveStructureObjClassId, testPrimitiveStructureObjIVs,  
testPrimitiveStructureObjKeywords, testPrimitiveStructureObjMethodDict, testPrimitiveStructureObjName, and testPrimitiveStructureObjSuperclassId.

Vous pouvez les exécuter en sélectionnant l'expression suivante (`RawObjTest selector : #testPrimitiveStructureObjClassId`) run. Notez que les tableaux commencent à 1 dans Pharo. Vous trouverez ci-dessous la liste des primitives que vous devez implémenter.

Implémentez dans le protocole 'object structure primitives' les primitives qui gèrent :

- la classe de l'instance représentée par un symbole. `objClassId`, `objClassId` : `aSymbol`. Le receveur est un `objObject`. Cela signifie que cette primitive peut être appliquée sur n'importe quel `objInstance`

pour obtenir son identifiant de classe.

Implémenter dans le protocole 'class structure primitives' les primitives qui gèrent :

- le nom de la classe : `objName`, `objName` : `aSymbol`. Le receveur est un `objClass`.
- la superclasse : `objSuperclassId`, `objSuperclassId` : `aSymbol`. Le receveur est un `objClass`.
- les variables d'instance : `objIVs`, `objIVs` : `anOrderedCollection`. Le receveur est un `objClass`.
- la liste des mots-clés : `objKeywords`, `objKeywords` : `anOrderedCollection`. Le receveur est un `objClass`.
- le dictionnaire de méthodes : `objMethodDict`, `objMethodDict` : `anIdentityDictionary`. Le receveur est un `objClass`.

## 2.8 Trouver la classe d'un objet

Chaque objet conserve l'identifiant de sa classe (son nom). Par exemple, une instance de `ObjPoint` a alors la structure suivante : `#(#ObjPoint 10 15)` où `#ObjPoint` est un symbole identifiant la classe `ObjPoint`.

### Votre travail.

En utilisant la primitive `giveClassNamed` : `aSymbol` définie au niveau de la classe `Obj`, définissez la primitive `objClass` dans le protocole 'object-structure primitive' qui renvoie l'`objInstance` qui représente sa classe (les classes sont aussi des objets dans `ObjVLisp`).

Assurez-vous que vous exécutez la méthode de test : `testClassAccess`

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

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

Nous sommes maintenant prêts à manipuler les `objInstances` via l'API appropriée. Nous allons maintenant utiliser la classe `ObjTest` pour des tests plus élaborés.

## 2.9 Accès aux valeurs des variables d'instance des objets

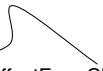
### Une première méthode simple.

Le test suivant illustre le comportement du message `offsetFromClassOfInstanceVariable:`.

```

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

```




**Figure 2-3** Décalage de la variable d'instance demandé à la classe.

```

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

```



**Figure 2-4** L'offset de la variable d'instance demandé à l'instance elle-même.

```

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

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

```

## Votre travail.

Dans le protocole 'iv management' définissez une méthode appelée `offsetFromClassOfInstanceVariable` : `aSymbol` qui renvoie le décalage de la variable d'instance représentée par le symbole donné en paramètre. Elle renvoie 0 si la variable n'est pas définie. Consultez les tests `#testIVOffset` de la classe `ObjTest`.

Conseils : Utilisez la méthode Pharo `indexOf`. Faites attention à ce qu'une telle primitive soit appliquée à une `objClass` comme indiqué dans le test.

Assurez-vous que vous exécutez la méthode de test : `testIVOffset`.

## Une deuxième méthode simple.

Le test suivant illustre le comportement attendu

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

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

## Votre travail.

En utilisant la méthode précédente, définissez dans le protocole 'iv management' :

1. la méthode `offsetFromObjectOfInstanceVariable` : aSymbol qui retourne le décalage de la variable d'instance. Notez que cette fois-ci la méthode est appliquée à une objInstance présentant une instance et non une classe (comme le montre la figure 2-4).
2. la méthode `valueOfInstanceVariable` : aSymbol qui retourne la valeur de cette variable d'instance dans l'objet donné comme le montre le test ci-dessous.

Notez que pour la méthode `offsetFromObjectOfInstanceVariable`, vous pouvez vérifier que la variable d'instance existe dans la classe de l'objet et, sinon, soulever une erreur en utilisant la méthode `Pharo error` :.

Assurez-vous que vous exécutez la méthode de test : `testIVOffsetAndValue` et qu'elle passe.

## 2.10 Allocation et initialisation des objets

La création d'un objet est la composition de deux opérations élémentaires : son *allocation* et son *initialisation*. Nous définissons maintenant les primitives qui nous permettent d'allouer et d'initialiser un objet. Rappelons que

- l'allocation est une méthode de classe qui retourne une structure presque vide, presque vide car l'instance représentée par la structure doit au moins connaître sa classe, et que
- l'initialisation est une méthode d'instance qui, étant donné une instance nouvellement allouée et une liste d'arguments d'initialisation, remplit l'instance.

### Allocation d'instance

Comme le montre la classe `ObjTest`, si la classe `ObjPoint` possède deux variables d'instance : `ObjPoint allocateAnInstance` renvoie `new(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)
```

### Votre travail.

Dans le protocole 'instance allocation' implémentez la primitive appelée `allocateAnInstance` qui envoyée à un *objClass* retourne une nouvelle instance dont les valeurs des variables d'instance sont nil et dont l'*objClassId* représente l'*objClass*.

Assurez-vous que vous exécutez la méthode de test : `testAllocate`.

## 2.11 Primitives de mots-clés

L'implémentation originale d'ObjVLisp utilise la facilité offerte par les mots-clés Lisp pour faciliter la spécification des valeurs des variables d'instance pendant la création de l'instance. Elle fournit également un moyen uniforme et unique de créer des objets. Nous devons implémenter certaines fonctionnalités pour supporter les mots-clés. Cependant, , comme ce n'est pas vraiment intéressant et que vous perdez du temps, nous vous donnons toutes les primitives nécessaires.

### Votre travail.

Toutes les fonctionnalités pour gérer les mots-clés sont définies dans le protocole 'keyword management'. Lisez le code et le test associé appelé `testKeywords` dans la classe `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))

```

Assurez-vous que vous exécutez la méthode de test : `testKeywords` et qu'elle passe.

## 2.12 Initialisation des objets

Une fois qu'un objet est alloué, il peut être initialisé par le programmeur en spécifiant une liste de valeurs d'initialisation. On peut représenter une telle liste par un tableau contenant alternativement un mot-clé et une valeur comme `#(#toto 33 #x 23)` où 33 est associé à `#toto` et 23 à `#x`.

### Votre travail.

Lire dans le protocole '`instance initialization`' la primitive `initializeUsing : anArray` qui, lorsqu'elle est envoyée à un objet avec une liste d'initialisation, renvoie l'objet initialisé.

```

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

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

```

## 2.13 Héritage statique des variables d'instance

Les variables d'instance sont héritées statiquement au moment de la création de la classe. La forme la plus simple de l'héritage des variables d'instance est de définir l'ensemble complet des variables d'instance comme la *fusion ordonnée* entre les variables d'instance héritées et les variables d'instance définies localement. Pour des raisons de simplicité et de similitude avec la plupart des langages, nous avons choisi d'interdire les variables d'instance dupliquées dans la chaîne d'héritage.

## Votre travail.

Dans le protocole 'iv inheritance', lisez et comprenez la primitive `computeNewIVFrom : superIVOrdCol with : localIVOrdCol`.

La primitive prend deux collections ordonnées de symboles et retourne une collection ordonnée contenant l'union des deux collections ordonnées mais avec la contrainte supplémentaire que l'ordre des éléments de la première collection ordonnée est conservé. Regardez la méthode de test `testInstanceVariableInheritance` ci-dessous pour des exemples.

Assurez-vous que vous exécutez la méthode de test : `testInstanceVariableInheritance` et qu'elle passe.

```
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)
```

## Remarque annexe

On pourrait penser que garder le même ordre des variables d'instance entre une superclasse et sa sous-classe n'est pas un problème. C'est en partie vrai dans cette implémentation simple car les accesseurs de variables d'instance calculent à chaque fois le décalage correspondant pour accéder à une variable d'instance en utilisant la primitive `offsetFromClassOfInstanceVariable:`. Cependant, la structure (ordre des variables d'instance) d'une classe est codée en dur par les primitives. C'est pourquoi votre implémentation de la primitive `computeNewIVFrom:with:` doit prendre en charge cet aspect.

## 2.14 Gestion des méthodes

Une classe stocke le comportement (exprimé par des méthodes) partagé par toutes ses instances dans un dictionnaire de méthodes. Dans notre implémentation, nous représentons les méthodes en associant un symbole à un *bloc* Pharo, une sorte de méthode anonyme. Le bloc est ensuite stocké dans le dictionnaire de méthodes d'une `objClass`.

Dans cette implémentation, nous n'offrons pas la possibilité d'accéder directement aux variables d'instance de la classe dans laquelle la méthode est

définie. Cela pourrait être fait en partageant un environnement commun entre toutes les méthodes. Le programmeur doit utiliser des accesseurs ou les méthodes `setIV` et `getIV` `objMethods` définies sur `ObjObject` pour accéder aux variables d'instance. Vous pouvez trouver la définition de ces méthodes dans le protocole d'amorçage du côté de la classe de `Obj`.

Dans notre implémentation d'`ObjVLisp`, nous n'avons pas de syntaxe pour le passage de messages. Au lieu de cela, nous appelons les primitives en utilisant la syntaxe Pharo pour le passage de messages (en utilisant le message `send:withArguments:`). L'expression `objself getIV : x` est exprimée en `ObjVLisp` comme suit `objself send: #getIV withArguments: #(x)`.

Le code suivant décrit la définition de la méthode accesseur `x` définie sur l'objClass `ObjPoint` qui invoque un accès au champ en utilisant le message `getIV`.

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

En première approximation, ce code créera le bloc suivant qui sera stocké dans le dictionnaire des méthodes de la classe: `[ :objself | objself send: #getIV withArguments: #(x) ]`. Comme vous pouvez le remarquer, dans notre implémentation, le receveur est toujours un argument explicite de la méthode. Ici, nous l'avons nommé `objself`.

## Définition d'une méthode et envoi d'un message

Comme nous voulons garder cette implémentation aussi simple que possible, nous ne définissons qu'une seule primitive pour l'envoi d'un message : il s'agit de `send:withArguments:`. Pour voir la correspondance entre les manières Pharo et `ObjVLisp` d'exprimer l'envoi d'un message, regardez la comparaison ci-dessous :

```
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)
```

Alors qu'en Pharo, vous écririez la définition de méthode suivante :

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

Dans notre implémentation de `ObjVLisp` vous écrivez :

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

## Votre travail.

Nous fournissons toutes les primitives qui gèrent la définition des méthodes. Dans le protocole 'gestion des méthodes' regardez les méthodes `addMethod : aSelector args : aString withBody : aStringBlock`, `removeMethod : aSelector` et `doesUnderstand : aSelector`. Mettez en œuvre `bodyOfMethod : aSelector`.

Assurez-vous que vous exécutez la méthode de test : `testMethodManagement`

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

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

## 2.15 Passage de messages et recherche dynamique

L'envoi d'un message est le résultat de la composition de la *recherche de méthode* et de *l'exécution*. La primitive `basicSend:withArguments:from:` suivante ne fait que l'implémenter. Tout d'abord, elle recherche la méthode dans la classe ou la superclasse du receveur, si la méthode a été trouvée, elle l'exécute, sinon `lookup:` retourne nil et nous levons une erreur `Pharo`.

```
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' ]

```

Sur la base de cette primitive, nous pouvons exprimer `send:withArguments:` comme suit :

```

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

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

```

## 2.16 Recherche de méthode

La primitive `lookup : selector` appliquée à une `objClass` doit retourner la méthode associée au sélecteur si elle l'a trouvée, sinon `nil` pour indiquer qu'elle a échoué.

### Votre travail.

Implémenter la primitive `lookup : selector` qui, envoyée à un `objClass` avec un sélecteur de méthode, un symbole et le receveur initial du message, retourne le corps de la méthode associée au sélecteur dans l'`objClass` ou ses superclasses. De plus, si la méthode n'est pas trouvée, `nil` est retourné.

Assurez-vous d'exécuter les méthodes de test : `testNilWhenErrorInLookup` et `testRaisesErrorSendWhenErrorInLookup` dont le code est donné ci-dessous :

```

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"

```





## Gestion de super

Pour invoquer une méthode cachée d'une superclasse, en Java et Pharo vous utilisez `super`, ce qui signifie que la recherche commencera au-dessus de la classe définissant la méthode contenant l'expression `super`. En fait, nous pouvons considérer qu'en Java ou Pharo, `super` est un sucre syntaxique pour se référer au receveur mais en changeant l'endroit où la recherche de la méthode commence. C'est ce que nous voyons dans notre implémentation où nous n'avons pas de support syntaxique.

Voyons comment nous allons exprimer la situation suivante.

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

Dans notre implémentation de `ObjVlisp`, nous n'avons pas de construction syntaxique pour exprimer `super`, vous devez utiliser le message `super:with-Arguments:` de Pharo comme suit.

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

Notez que `superClassOfClassDefiningTheMethod` est une variable liée à la superclasse de `anObjClass`, c'est-à-dire la classe définissant la méthode `bar` (voir plus loin).

```
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
```

### 3.1 Représentation de super

Nous aimerions vous expliquer d'où vient la variable `superClassOfClassDefiningTheMethod`. Lorsque nous comparons la primitive `send:withArguments:`, pour les super envois nous avons ajouté un troisième paramètre à la primitive et nous l'avons appelé `super:withArguments:from:`.

Ce paramètre supplémentaire correspond à la superclasse de la classe dans laquelle la méthode est définie. Cet argument doit toujours avoir le même nom, c'est-à-dire `superClassOfClassDefiningTheMethod`. Cette variable sera liée lorsque la méthode sera ajoutée dans le dictionnaire des méthodes d'une `objClass`.

Si vous voulez comprendre comment nous lions la variable, voici l'explication : En fait, une méthode n'est pas seulement un bloc mais elle a besoin de connaître la classe qui la définit ou sa superclasse. Nous avons ajouté cette information en utilisant la curryfication. (une curryfication est la transformation d'une fonction à  $n$  arguments en fonction avec moins d'argument mais une capture d'environnement :  $f(x, y) = (+ x y)$  est transformée en une fonction  $f(x) = f(y)(+ x y)$  qui retourne une fonction d'un seul argument  $y$  et où  $x$  est lié à une valeur et obtient un générateur de fonction). Par exemple,  $f(2, y)$  renvoie une fonction  $f(y) = (+ 2 y)$  qui ajoute son paramètre à 2. Une curryfication agit comme un générateur de fonction où l'un des arguments de la fonction originale est fixé.

Dans Pharo, nous enveloppons le bloc représentant la méthode autour d'un autre bloc avec un seul paramètre et nous lions ce paramètre avec la superclasse de la classe définissant la méthode. Lorsque la méthode est ajoutée au dictionnaire des méthodes, nous évaluons le premier bloc avec la superclasse comme paramètre, comme illustré ci-dessous :

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

Vous savez donc maintenant d'où vient la variable `superClassOfClassDefiningTheMethod`. Assurez-vous que vous exécutez la méthode de test : `testMethodLookup` et qu'elle passe.



## Votre travail.

Vous devriez maintenant mettre en œuvre `super : selector withArguments : arguments from : aSuperclass` en utilisant la primitive `basicSend:withArguments:from:.`

## 3.2 Gestion des messages non compris

Nous pouvons maintenant revoir la gestion des erreurs. Au lieu de générer une erreur Pharo, nous voulons envoyer un message `ObjVlisp` au destinataire du message pour lui donner une chance de détecter l'erreur.

Comparez les deux versions suivantes de `basicSend : selector withArguments : arguments from : aClass` et proposez une implémentation de `sendError : selector withArgs : arguments.`

```
Obj >> basicSend: selector withArguments: arguments from: aClass
| 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
| methodOrNil |
methodOrNil := (aClass lookup: selector).
^ methodOrNil
    ifNotNil: [ methodOrNil valueWithArguments: (Array with: self)
, arguments ]
    ifNil: [ self sendError: selector withArgs: arguments ]
```

Il convient de noter que la méthode `objVlisp` est définie comme suit dans la classe `ObjObject` (voir la méthode `bootstrap` du côté de la classe `Obj`). La méthode `obj error` attend un seul paramètre : un tableau d'arguments dont le premier élément est le sélecteur du message non compris.

```
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:
```

```
i      selector))  
L
```

Assurez-vous de lire et d'exécuter la méthode de test : `testSendErrorRaisesErrorSendWhenErrorInLookup`. Regardez l'implémentation de la méthode `#error` définie dans `ObjObject` et dans `assembleObjectClass` de la classe `ObjTest`.



## Amorçage du système

Maintenant que vous avez implémenté tous les comportements dont nous avons besoin, vous êtes prêt à amorcer le système : cela signifie créer le noyau constitué des classes `ObjObject` et `ObjClass` à partir d'elles-mêmes. L'idée d'un bootstrap intelligent est d'être aussi paresseux que possible et d'utiliser le système pour se créer lui-même en créant rapidement une première classe fausse mais fonctionnelle avec laquelle nous construirons le reste.

Trois étapes composent le bootstrap `ObjVlisp`,

1. nous créons à la main la partie minimale de l'objClass `ObjClass` et ensuite
2. nous l'utilisons pour créer normalement `ObjObject` `objClass` et ensuite
3. nous recréons normalement et complètement `ObjClass`.

Ces trois étapes sont décrites par la méthode bootstrap suivante de la classe `Obj`. Notez que le bootstrap est défini comme les méthodes de la classe `Obj`.

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

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

Pour vous aider à implémenter la fonctionnalité des classes d'objets `ObjClass` et `ObjObject`, nous avons défini un autre ensemble de tests dans la classe `ObjTestBootstrap`. Lisez-les.

## 4.1 Création manuelle de la classe ObjClass

La première étape consiste à créer manuellement la classe ObjClass. Par manuellement, nous entendons créer un tableau (car nous avons choisi un tableau pour représenter les instances et les classes en particulier) qui représente l'objClass ObjClass, puis définir ses méthodes. Vous implémenterez/liserez ceci dans la primitive manuallyCreateObjClass comme indiqué ci-dessous :

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

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

Pour cela, vous devez implémenter/lire toutes les primitives qui le composent.

### Votre travail.

Au niveau de la classe dans le protocole 'bootstrap objClass manual' lire ou implémenter : la primitive manualObjClassStructure qui renvoie un objObject qui représente la classe ObjClass.

Assurez-vous d'exécuter la méthode de test : testManuallyCreateObjClassStructure.

- Comme le initialize de cette première phase du bootstrap n'est pas facile nous vous donnons son code. Notez que la définition de l'objMethod initialize est faite dans la méthode primitive 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'
```

Notez que cette méthode fonctionne sans héritage puisque la classe `ObjObject` n'existe pas encore.

La primitive `defineAllocateMethodIn` : `anObjClass` définit dans un `ObjClass` passé en argument la méthode `allocate`. `allocate` ne prend qu'un seul argument : la classe pour laquelle une nouvelle instance est créée comme indiqué ci-dessous :

```
defineAllocateMethodIn: class

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

Suivant le même principe, définissez la primitive `defineNewMethodIn` : `anObjClass` qui définit dans une `ObjClass` passée en argument l'objMethod `new`. `new` prend deux arguments : une classe et une `initargs-list`. Elle doit invoquer les méthodes objMethod `allocate` et `initialize`.

### Votre travail.

Assurez-vous de lire et d'exécuter la méthode de test : `testManuallyCreateObjClassAllocate`.

### Remarques

Lisez attentivement les remarques ci-dessous et le code.

- Dans la méthode objMethod `manualObjClassStructure`, l'héritage des variables d'instance est simulé. En effet, le tableau de variables d'instance contient `#class` qui devrait normalement être hérité de `ObjObject` comme nous le verrons dans la troisième phase du bootstrap.
- Notez que la classe est déclarée dans le référentiel de classes à l'aide de la méthode `declareClass`.
- Notez que la méthode `#initialize` est une méthode de la métaclasse `ObjClass` : lorsque vous créez une classe, la méthode `initialize` est in-

voquée sur une classe ! La méthode `initialize` définie sur la méta-classe `ObjClass` a deux aspects : le premier concerne l'initialisation de la classe comme toute autre instance (première ligne). Ce comportement est normalement réalisé en utilisant un super appel pour invoquer la méthode `initialize` définie dans `ObjObject`. La version finale de la méthode `initialize` le fera en utilisant `perform`. La seconde traite de l'initialisation des classes : effectuer l'héritage des variables d'instance, puis calculer les mots-clés de la classe nouvellement créée. Notez dans cette dernière étape que le tableau de mots-clés ne contient pas le mot-clé `#class` : car nous ne voulons pas laisser l'utilisateur modifier la classe d'un objet.

## 4.2 Création d'un `ObjObject`

Vous êtes maintenant dans la situation où vous pouvez créer la première classe réelle et normale du système : la classe `ObjObject`. Pour ce faire, vous envoyez le message `new` à la classe `ObjClass` en spécifiant que la classe que vous créez est nommée `#ObjObject` et que vous n'avez qu'une seule variable d'instance appelée `class`. Vous ajouterez ensuite les méthodes définissant le comportement partagé par tous les objets.

### Votre travail : `objObjectStructure`

Implémentez/lisez la primitive suivante `objObjectStructure` qui crée le `ObjObject` en invoquant le message `new` à la classe `ObjClass` :

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

La classe `ObjObject` est nommée `ObjObject`, possède une seule variable d'instance `class` et n'a pas de superclasse car elle est la racine du graphe d'héritage.

### Votre travail : `createObjObject`

Implémentez maintenant la primitive `createObjObject` qui appelle `objObjectStructure` pour obtenir le `objObject` représentant la classe `objObject` et y définir des méthodes. Pour vous aider, nous donnons ici le début d'une telle méthode

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

```
objObject addUnaryMethod: #isClass withBody: 'false'.  
objObject addUnaryMethod: #isMetaClass withBody: 'false'.  
...  
...  
^ objObject
```

Implémenter les méthodes suivantes dans ObjObject

- l'objMethod class qui, étant donné une objInstance, retourne sa classe (l'objInstance qui représente la classe).
- l'objMethod isClass qui renvoie false.
- l'objMethod isMetaClass qui renvoie false.
- l'objMethod error qui prend deux arguments : le receveur et le sélecteur de l'invocation originale et qui lève une erreur.
- l'objMethod getIV qui prend le receveur et un nom d'attribut, aSymbol, et renvoie sa valeur pour le receveur.
- L'objMethod setIV qui prend le receveur, un nom d'attribut et une valeur et définit la valeur de l'attribut donné à la valeur donnée.
- l'objMethod initialize qui prend le receveur et une initargs-list et initialise le receveur selon la spécification donnée par l'initargs-list. Notez qu'ici, la méthode initialize ne fait que remplir l'instance conformément à la spécification donnée par la liste d'initargs. Comparez avec la méthode initialize définie sur ObjClass.

Assurez-vous de lire et d'exécuter la méthode de test : testCreateObjObjectStructure.

Remarquez en particulier que cette classe n'implémente pas la méthode de classe new parce qu'elle n'est pas une métaclasse, mais qu'elle implémente la méthode d'instance initialize parce que tout objet doit être initialisé.

#### Votre travail : exécuter les tests

- Assurez-vous de lire et d'exécuter la méthode de test : testCreateObjObjectMessage.
- Assurez-vous que vous lisez et exécutez la méthode de test : testCreateObjObjectInstanceMessage.

### 4.3 Création de la classe ObjClass

En suivant la même démarche, vous pouvez maintenant recréer complètement la classe ObjClass. La primitive createObjClass est chargée de créer la classe finale ObjClass. Vous allez donc l'implémenter et définir toutes les

primitives dont elle a besoin. Maintenant nous ne définissons que ce qui est spécifique aux classes, le reste est hérité de la superclasse de la classe `ObjClass`, la classe `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
```

Pour que la méthode `createObjClass` fonctionne, nous devons implémenter la méthode qu'elle appelle. Implémenter alors

- la primitive `objClassStructure` qui crée la classe `ObjClass` en invoquant le message `new` à la classe `ObjClass`. Notez qu'au cours de cette méthode le symbole `ObjClass` fait référence à deux entités différentes car la nouvelle classe qui est créée en utilisant l'ancienne est déclarée dans le dictionnaire des classes avec le même nom.

## Votre travail.

Assurez-vous d'avoir lu et exécuté la méthode de test : `testCreateObjClassStructure`. Implémentez maintenant la primitive `createObjClass` qui commence comme suit :

```
Obj class >> createObjClass

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

Définissez également les méthodes suivantes :



- la méthode `objMethod isClass` qui renvoie `true`.
- la méthode `objMethod isMetaclass` qui renvoie `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: #()'.

```

- la primitive `defineInitializeMethodIn` : `anObjClass` qui ajoute l'`objMethod initialize` à l'`objClass` passée en argument. L'`objMethod initialize` prend le receveur (une `objClass`) et une liste d'`initargs` et initialise le receveur selon les spécifications données par la liste d'`initargs`. En particulier, il doit être initialisé comme n'importe quel autre objet, puis il doit calculer sa variable d'instance (c'est-à-dire que les variables d'instance héritées sont calculées), les mots-clés sont également calculés, le dictionnaire de méthodes doit être défini et la classe est ensuite déclarée comme une classe existante. Nous vous fournissons le modèle suivant pour vous aider.

```
Obj class>>defineInitializeMethodIn: objClass

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

```
Obj class >> defineInitializeMethodIn: objClass

objClass
  addMethod: #initialize
  args: 'initArray'
  withBody:
    'objself super: #initialize withArguments: {initArray}
    from: superClassOfClassDefiningTheMethod.
    :

```

```

objself objIVs: (objself
  computeNewIVFrom: (Obj giveClassName: objself
objSuperclassId) objIVs
  with: objself objIVs).
objself computeAndSetKeywords.
objself objMethodDict: IdentityDictionary new.
Obj declareClass: objself.
objself'

```

## Votre travail.

Assurez-vous que vous exécutez la méthode de test : `testCreateObjClassMessage`.

Notez les points suivants :

- Les variables d'instance spécifiées localement ne sont plus que les variables d'instance qui décrivent une classe. La variable d'instance `class` est héritée de `ObjObject`.
- La méthode `initialize` effectue maintenant un super envoi pour invoquer l'initialisation effectuée par `ObjObject`.

## 4.4 Premières classes d'utilisateurs : `ObjPoint`

Maintenant que `ObjVLisp` est créé, nous pouvons commencer à programmer quelques classes. Implémentez la classe `ObjPoint` et `ObjColoredPoint`. Voici une implémentation possible.

Vous pouvez choisir de l'implémenter au niveau de la classe `Obj` ou mieux encore dans une classe nommée `ObjPointTest`.

Veillez à ce que votre scénario couvre les aspects suivants :

- Créez tout d'abord la classe `ObjPoint`.
- Créez une instance de la classe `ObjPoint`.
- Envoyez à cette instance certains messages définis dans `ObjObject`.

Définissez la classe `ObjPoint` afin de pouvoir créer des points comme ci-dessous (créez une méthode `Pharo` pour la définir).

```

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).

```

Ajoutez ensuite quelques fonctionnalités à la classe ObjPoint comme les méthodes `x`, `x:`, `display` qui impriment le receveur.

```
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'.
```

Puis testez ces nouvelles fonctionnalités.

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

## 4.5 Premières classes d'utilisateurs : ObjColoredPoint

En suivant la même idée, définissez la classe ObjColored.

Créez une instance et envoyez-lui quelques messages de base.

```
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).
```

### Votre travail.

Définissez quelques fonctionnalités et invoquez-les : la méthode `color`, implémentez la méthode `display` de façon à ce qu'elle invoque la superclasse et ajoute quelques informations relatives à la couleur. Voici un exemple :

```
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: #()
```

## 4.6 Une métaclasse pour les premiers utilisateurs : ObjAbstract

Implémentez maintenant la métaclasse `ObjAbstract` qui définit les instances (classes) qui sont abstraites c'est-à-dire qui ne peuvent pas créer d'instances. Cette classe devrait lever une erreur lorsqu'elle exécute le message `new`.

L'exemple suivant vous montre une utilisation possible de cette métaclasse.

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

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

Vous devez redéfinir la méthode `new`. Notez que la `ObjAbstractClass` est une instance de `ObjClass` parce que c'est une classe et en hérite parce que c'est une métaclasse.

## 4.7 Nouvelles fonctionnalités que vous pouvez implémenter

Vous pouvez implémenter quelques fonctionnalités simples :

- définir une métaclasse qui définit automatiquement des accesseurs pour les variables des instances spécifiées.
- éviter que l'on puisse changer le sélecteur et les arguments lors de l'appel d'un super envoi.

### Variables partagées

Notez que contrairement à la proposition faite dans le 6ème postulat du modèle original d'ObjVLisp, les variables d'instance de classe ne sont pas équivalentes aux variables partagées. Selon le 6ème postulat, une variable partagée sera stockée dans l'instance représentant la classe et non dans une variable d'instance de la classe représentant les variables partagées. Par exemple si une station de travail a une variable partagée nommée `domaine`. Mais `domaine` ne devrait pas être une variable d'instance supplémentaire de la classe de station de travail. En effet, `domaine` n'a rien à voir avec la description de la classe.

La solution correcte est que `domain` est une valeur maintenue dans la liste de la variable partagée de la classe `Workstation`. Cela signifie que une *classe* possède une information supplémentaire pour la décrire : une variable d'instance `sharedVariable` contenant des paires. Nous devrions donc être en mesure d'écrire

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

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

Introduire les variables partagées : ajouter une nouvelle variable d'instance dans la classe `ObjClass`. classe `ObjClass` pour contenir un dictionnaire de liaisons de variables partagées (un symbole et une valeur) qui peut être interrogé à l'aide de méthodes spécifiques : `sharedVariableValue:`, `sharedVariableValue:put:`.





## Définitions choisies

Smith a été le premier à introduire la réflexion dans un langage de programmation avec 3Lisp [9]. Il définit la réflexion comme suit

- La capacité intégrale d'une entité à se représenter, à opérer et à se traiter de la même manière qu'elle représente, opère et traite son sujet principal.

Dans le contexte des protocoles de méta-objets, Bobrow [2] affine la définition comme suit :

- La réflexion est la capacité d'un programme à manipuler comme données quelque chose représentant l'état du programme pendant sa propre exécution. Il y a deux aspects d'une telle manipulation : *introspection* et *intercession* (...) Les deux aspects nécessitent un mécanisme pour encoder l'état d'exécution en tant que données ; fournir un tel encodage est appelé *réification*.

Maes a proposé quelques définitions pour la programmation réflexive [8]:

- Un *système informatique* est quelque chose qui *raisonne* sur et *agit* sur une partie du monde, appelée *domaine* du système.
- Un système de calcul peut également être "connecté de manière causale" à son domaine. Cela signifie que le système et son domaine sont liés de telle sorte que si l'un des deux change, cela entraîne un effet sur l'autre.
- Un *méta-système* est un système de calcul qui a pour domaine un autre système de calcul, appelé *système-objet*. (...) Un méta-système a une représentation de son système-objet dans ses données. Son programme spécifie le *méta-calcul* sur le système-objet et est donc appelé *méta-programme*.

- La *réflexion* est le processus de raisonnement et/ou d'action sur soi.
- Un *système réflexif* est un méta-système relié causalement qui a pour objet le système lui-même. Les données d'un système réflexif contiennent, en plus de la représentation d'une partie du monde extérieur, une représentation causalement connectée de lui-même, appelée *auto-représentation* du système. [...] Lorsqu'un système raisonne ou agit sur lui-même, on parle de *calcul réflexif*.
- Un langage avec une *architecture réflexive* est un langage dans lequel tous les systèmes ont accès à une représentation causale d'eux-mêmes.
- Un environnement de programmation a une *architecture de méta-niveau* s'il a une architecture qui supporte le méta-calcul, sans supporter le calcul réflexif.
- Le *méta-objet* d'un objet X représente les informations explicites sur X (par exemple, sur son comportement et son implémentation). L'objet X lui-même regroupe les informations sur l'entité du domaine qu'il représente.



# Bibliography

- [1] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.
- [2] D. G. Bobrow, R. P. Gabriel, and J. White. CLOS in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.
- [3] N. Bouraqadi, T. Ledoux, and F. Rivard. Safe metaclass programming. In *Proceedings OOPSLA '98*, pages 84–96, 1998.
- [4] P. Cointe. Metaclasses are first class: the ObjVlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 156–167, Dec. 1987.
- [5] S. Danforth and I. R. Forman. Derived metaclass in SOM. In *Proceedings of TOOLS EUROPE '94*, pages 63–73, 1994.
- [6] N. Graube. Metaclass compatibility. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 305–316, Oct. 1989.
- [7] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [8] P. Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Belgium, Jan. 1987.
- [9] B. C. Smith. Reflection and semantics in lisp. In *Proceedings of POPL '84*, pages 23–3, 1984.
- [10] G. L. Steele. *Common Lisp The Language*. Digital Press, second edition, 1990.