Apprendre les concepts objets avec Java

Plan du séminaire

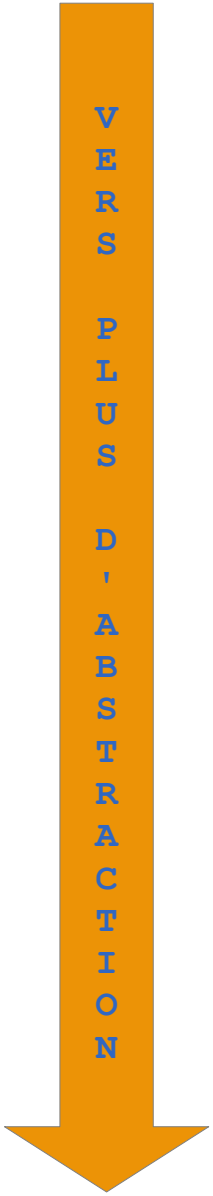
Plan Général

- Introduction
- La POO (Programmation Orientée Objet)
- Fondement n°1 : l'encapsulation
- Fondement n°2 : la communication par message
- Fondement n°3 : le polymorphisme
- Fondement n°4 : les classes (ou les prototypes)
- Fondement n°5 : l'héritage
- Fondement n°6 : l'abstraction
- Notions avancées
- Conclusion

Introduction

Evolution des langages de programmation

- **Langage machine**
  - Codage en hexadécimal des instructions et des données
- **Assembleur**
  - Des instructions un peu plus lisibles (move, jump, …)
  - Très proche de la machine (notions de registre, d'adresse mémoire, ...)
- **Programmation structurée**
  - Élimination des GOTO au profit des procédures (sous-routines)
  - Le programme est structuré par une « décomposition fonctionnelle »
- **Programmation fonctionnelle**
  - Théorie de l'information : tout est fonction, constituée d'autres fonctions
  - Très puissant mais très théorique avec des concepts très abstraits
- **Programmation objet / logique / L4G**
  - Décrire ce que l'on veut faire plutôt que comment le faire



Programmation « classique »

- **Les évolutions ne sont pas aussi marquées**
  - Les langages évoluent et incorporent souvent plusieurs paradigmes
- **En général, on retrouve :**
  - La notion de **`procédure`**/**`fonction`**/**`sous-routine`** pour organiser son code
  - La notion de **`variable`** pour organiser ses données
  - Un **`point d'entrée`** dans le programme (souvent une fonction appelée **`main`**)
  - Les instructions de la programmation structurée :
    - Instruction de **`branchement conditionnel`** (si-alors-sinon, souvent appelée **`if`**)
    - Instruction de **`boucle`** (répéter jusqu'à - tant que, faire - pour - souvent appelées **`while`** et **`for`**)
  - Ainsi que la notion de **`structure de données`** (définition de types complexes)
  - La notion de **`module`** et/ou de **`bibliothèque`**
    - Regroupement de fonctions et de structures de données autour d'un thème
    - Introduit généralement une notation pointée (module.fonction())

La POO
(Programmation Orientée Objet)

Les motivations pour un nouveau paradigme

- **La programmation modulaire a permis un grand bond**
  - Les programmes sont devenus beaucoup plus lisibles et simples à maintenir
  - Des bibliothèques de plus en plus riches ont facilité la vie des développeurs
- **La puissance des machines a accompagné cette progression**
  - Des programmes de plus en plus complexes sont devenus envisageables
  - Le nombre de ligne de code augmentant, il a fallu travailler en équipe
  - La lisibilité et la maintenabilité est devenue encore plus importante
- **Tout ceci a conduit à mettre au point des méthodes de travail**
  - Les projets informatiques sont dorénavant découpés en phases
    - Analyse, Conception, Programmation, Test, Intégration, Recette, ...
  - Des méthodes telles que **MERISE** industrialisent alors le développement logiciel
    - Beaucoup utilisé dans les années 70-90, encore partiellement présent aujourd'hui
- **Mais de nouveaux problèmes sont apparus**

Les motivations pour un nouveau paradigme

- **Un des problèmes les plus délicats est la cohérence des données**
  - Une problématique de données mais nécessairement gérée par des traitements
- **Exemple du système d'agenda contenant des RendezVous**
  - Comment garantir que toutes les heures seront correctes ? (1087 est invalide)
  - Comment garantir que tout RendezVous vérifiera **`Debut < Fin`** ?

```
SUB ModifierDebut (r AS RendezVous, heure AS INTEGER)
  IF VerifierHeure(heure) THEN
    IF heure < r.Fin THEN
      r.Debut = heure
    ELSE
      PRINT "Heure debut invalide"
    END IF
  ELSE
    PRINT "Heure invalide"
  END IF
END SUB
```

```
TYPE RendezVous
  Description AS STRING * 32
  Lieu AS STRING * 20
  Date AS LONG
  Debut AS INTEGER
  Fin AS INTEGER
END TYPE

CONST MAX = 100
DIM Agenda(MAX) AS RendezVous
```

Date sous forme d'un entier AAAAMMJJ

Heure sous forme d'un entier HHMM

```
Agenda(1).Debut = 1087
Agenda(1).Fin = 1030
```

Comment obliger à utiliser cette procédure ?

Comment interdire ce type de code (heure début invalide et après heure fin ?

Fondement n°1
L'encapsulation

Principes « objet »

- **La programmation orientée objet (POO) est d'abord conceptuelle**
  - On cherche à diminuer le fossé entre informaticiens et non informaticiens
  - La technique doit s'effacer au service de la compréhension et la communication
  - On veut gagner en abstraction en s'éloignant encore plus de la technique
  - On veut se rapprocher de la manière humaine normale de penser
- **Le mot « objet » fait référence aux choses qui nous entourent**
  - Tout ce que nous manipulons au quotidien sont des « objets »
  - Souvent fabriqués mais tout ce qui affecte nos sens peut être appelé « objet »
- **Le mot « objet » prend aussi le sens de « sujet »**
  - Quand on parle de « l'objet de la discussion » par exemple
- **L'intérêt n°1 d'un objet c'est qu'on peut le nommer et le décrire**
  - En terme de **`caractéristiques`** : ses propriétés ou attributs
  - En terme de **`comportements`** : ce qu'il sait faire, ce qu'on peut en faire

Réification

- **Un mot savant pour briller en société**
  - **`Réification`** : « fait de transformer en chose, ce qui est une idée, un mouvement, un concept, par exemple la conscience » (https://fr.wiktionary.org/wiki)
  - On peut traduire le verbe « **`réifier`** » par « **`chosifier`** »
- **C'est fondamentalement le travail de l'analyste – concepteur**
  - Tout concept cité par l'utilisateur sera traduit par un objet
  - On se concentrera pour se mettre d'accord sur son nom
  - Et ensuite pour le décrire en terme de propriétés et de comportements
- **Exemple**
  - Un agenda permet de noter les rendez-vous
  - Un rendez-vous comporte description, lieu, date, heure de début et de fin
  - Un rendez-vous peut être avancé, retardé, annulé
  - L'agenda doit vérifier que les rendez-vous ne se chevauchent pas
  - L'agenda doit proposer des créneaux libres pour un futur rendez-vous

Fondements

- **L'approche conceptuelle donne lieu à plusieurs `fondements`**
  - Les principes de base que l'on doit retrouver dans tout langage orienté objet
  - La plupart de ces fondements viennent d'une logique de la vie courante
  - Ils visent à résoudre les problèmes rencontrés avec la programmation classique
    - Améliorer dans les programmes la lisibilité, la maintenabilité, la robustesse, l'évolutivité, etc...
    - En particulier lorsqu'ils atteignent plusieurs centaines de milliers de lignes de code
- **Un des problèmes évoqués est la séparation données/traitements**
  - **`Modèle traditionnel`** : fonde la conception sur ces **`deux aspects séparés`**
  - **`Modèle objet`** : propose de décomposer le logiciel en **`un seul aspect`**
    - Des entités collaboratives appelées « objets » et réunissant données et traitements
- **Dans une certaine mesure : `objet = données + traitements`**
  - C'est une proposition partiellement correcte mais elle a le mérite d'être simple

Fondement 1 : l'encapsulation

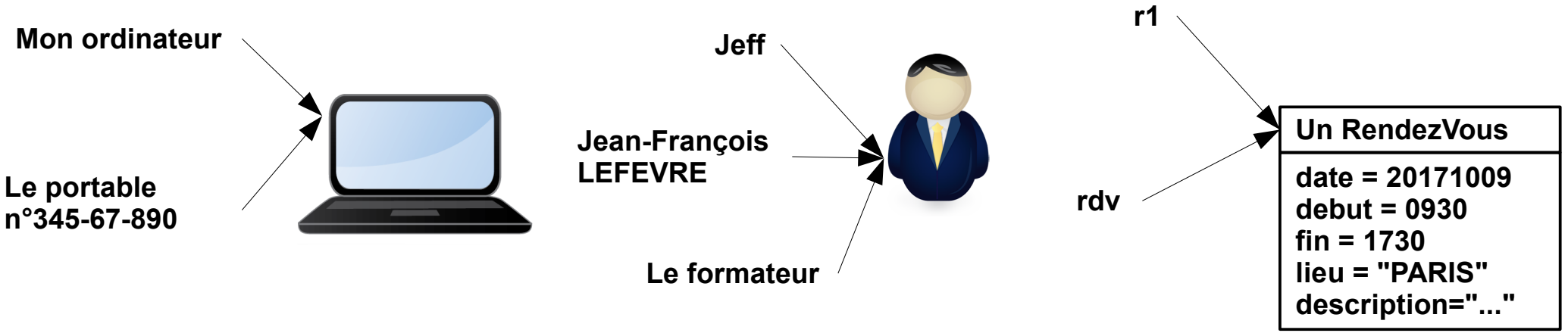
- **Le premier fondement est `l'encapsulation`**
  - Un objet est une **`capsule hermétique`** contenant données et traitements
- **Ce n'est finalement que le principe de modularité**
  - La notion de module vue précédemment : un objet est une sorte de module
  - L'objet est une boite noire dont l'intérieur n'a pas a être connu pour l'utiliser
- **C'est aussi le principe ordinaire de la vie quotidienne**
  - Nos objets courants sont généralement délimités par un boîtier
  - Ce boîtier permet **`d'assurer l'intégrité de l'objet`**
    - On n'est pas censé aller trifouiller à l'intérieur de la télévision pour l'utiliser
    - Si on le fait, la garantie du fabriquant saute

Un RendezVous
date = 20171009 debut = 0930 fin = 1730 lieu = "PARIS" description="..."

- **Les données se retrouvent ainsi correctement isolées**
  - Elles sont détenues dans des variables internes : les **`attributs`** de l'objet
  - Elles sont ainsi rendues inaccessibles de l'extérieur de l'objet

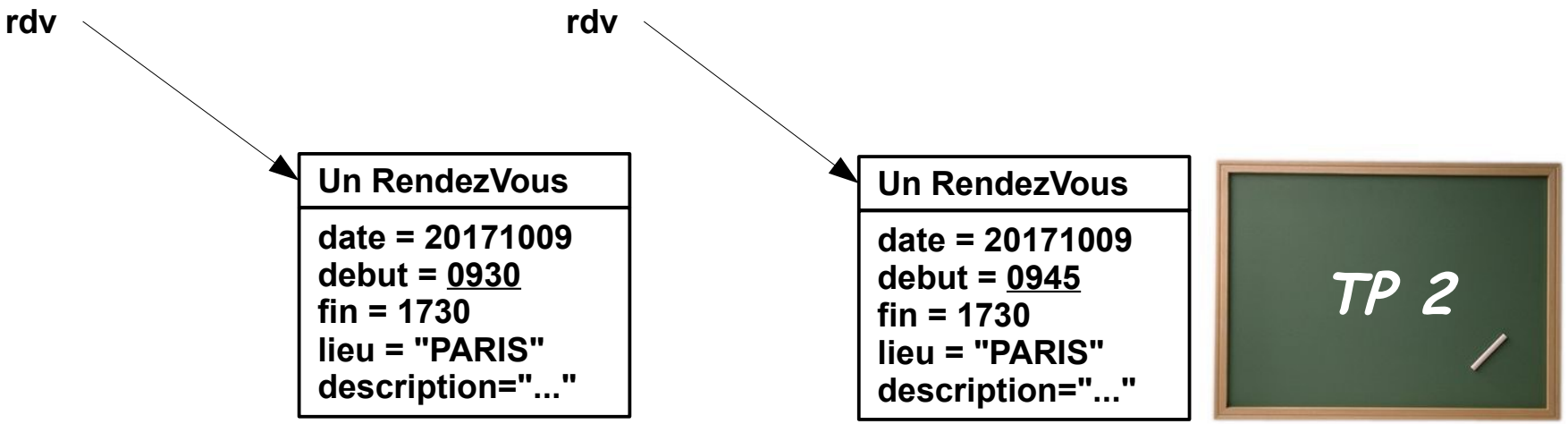
Identité

- **Un objet possède une `identité` intrinsèque**
  - Correspond à l'idée qu'on peut montrer un objet sans le confondre avec un autre
  - Techniquement, ce sera son adresse en mémoire, définie par le système
- **La notion de `variable` change un peu**
  - Une variable devient un **`nom`** local permettant de **`référencer`** un objet
  - **Remarque :** plusieurs variables peuvent référencer le même objet
- **Exemple : `r1` et `rdv` : deux variables référençant le même objet**



État

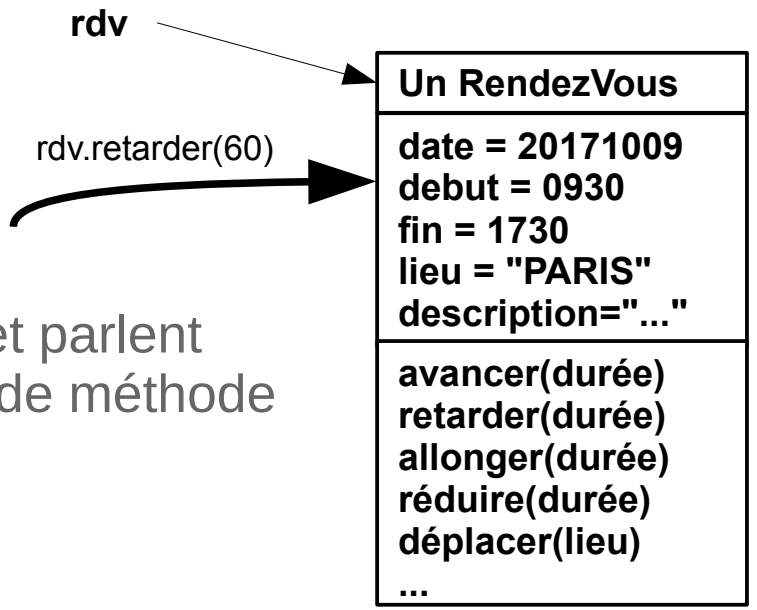
- **Les valeurs des attributs constituent `l'état de l'objet`**
  - On dira que l'objet a changé d'état si un attribut change de valeur
  - Correspond à la notion habituelle d'état d'un appareil par exemple
  - Exemple : une télévision peut être allumée, éteinte ou en veille
- **Certains objets ne changent pas d'état**
  - Ils sont alors qualifiés de **`immuable`** (immutable)
- **Exemple**
  - L'objet rendez-vous identifié par **`rdv`** a changé d'état (démarre 15mn plus tard)



Fondement n°2
La communication par message

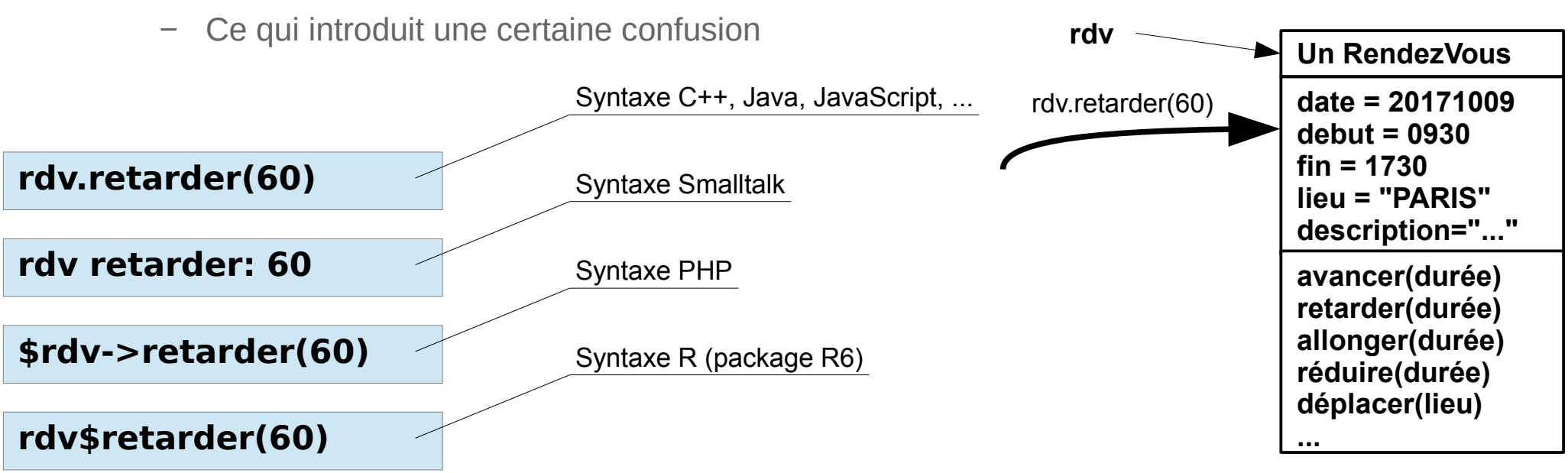
Fondement 2 : message/méthode

- **L'objet encapsule aussi des traitements : ses `méthodes`**
  - Des procédures ou fonctions internes, appelés **`opérations`** dans UML
  - Ce sont les seuls traitements capables d'accéder directement aux attributs
    - Car ils sont aussi à l'intérieur de l'objet
  - Si elles sont encapsulées, comment les activer depuis l'extérieur de l'objet ?
- **L'activation d'une méthode se fait par l'envoi d'un `message`**
  - On s'adresse à un **`objet`** en lui envoyant un **`message`**
  - Pour **`activer`** une de ses **`méthodes`**
- **Remarque**
  - Le mot « message » n'est pas toujours utilisé
  - Certains développeurs ne font pas la distinction et parlent abusivement d'appel de fonction ou d'invocation de méthode ce qui n'est très précis



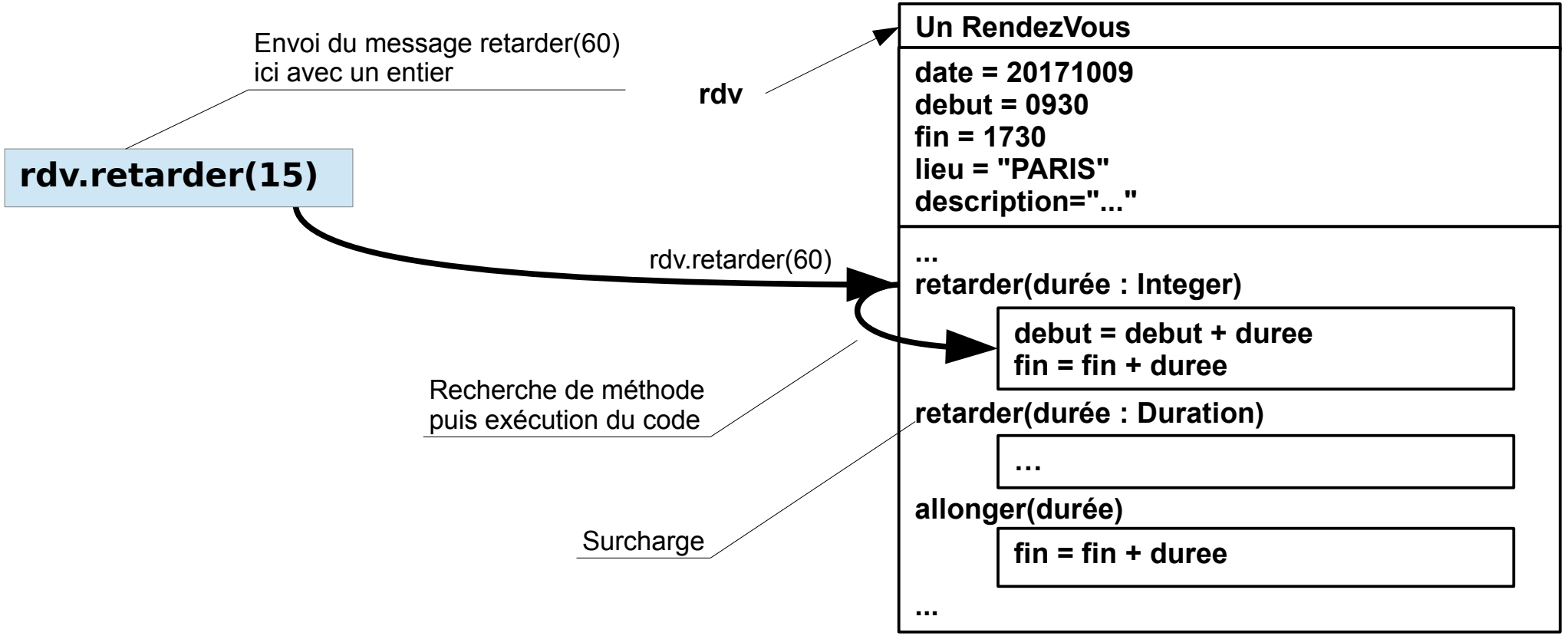
Envoi de message

- **L'envoi de message `n'est pas` un appel de fonction**
  - Une fonction est simplement invoquée par son nom
  - Avec l'envoi de message, il faut aussi indiquer le **`destinataire`** du message
- **La syntaxe dépend du langage**
  - Mais il faut toujours indiquer le **`receveur du message`**
    - C'est-à-dire l'objet auquel on s'adresse
  - Certains langages comme Java introduisent un receveur par défaut (implicite)
    - Ce qui introduit une certaine confusion



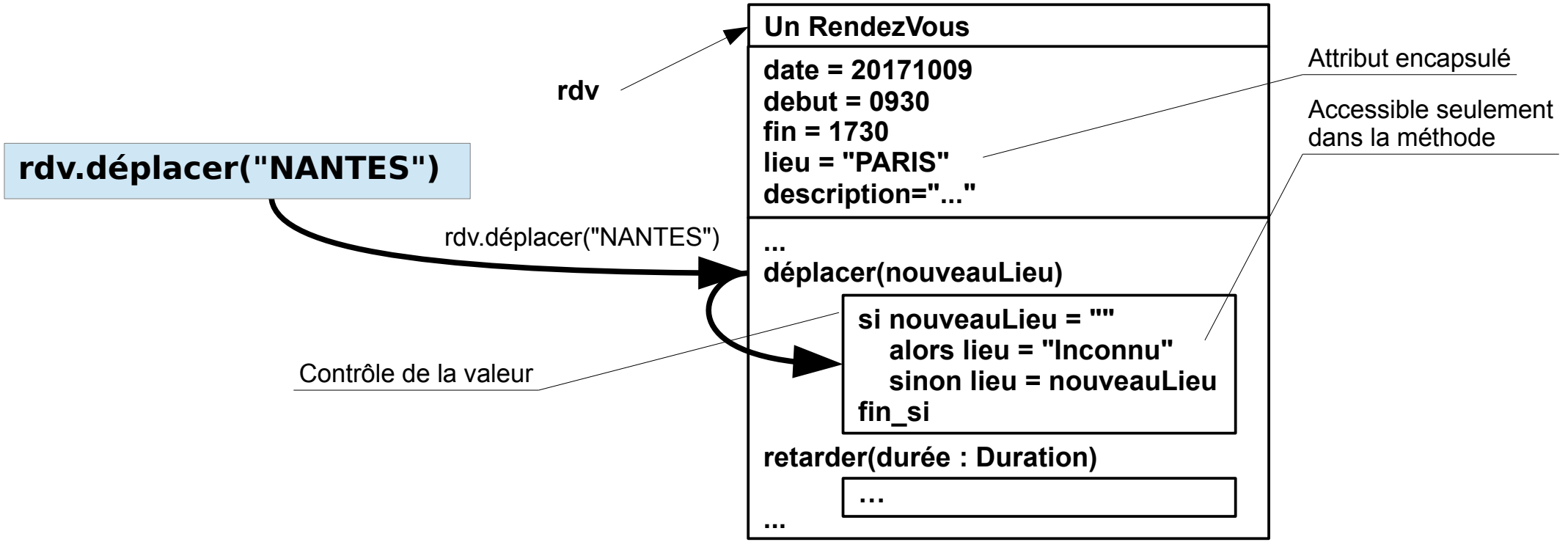
Recherche de méthode

- **Lorsqu'un objet reçoit un message, il `recherche` la méthode**
  - Évidemment, la **`correspondance`** se base sur le **`nom`**
  - Les langages typés autorisent généralement la **`surcharge`**
    - Lorsque des méthodes ont le même nom avec des paramètres différents
    - On parle de **`signature`** pour désigner le nom, le nombre et les types des paramètres



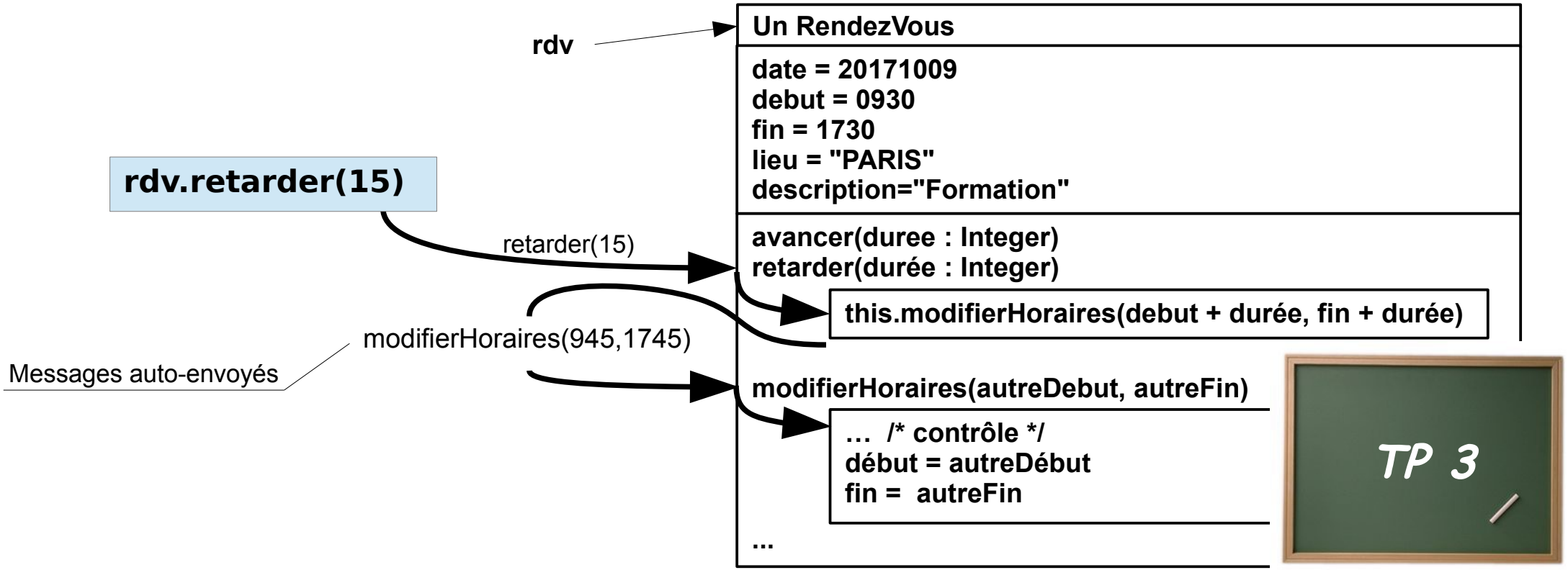
Encapsulation = garantie de cohérence

- **Les attributs sont censés être privés, inaccessibles de l'extérieur**
  - L'objet peut ainsi **`garantir la cohérence`** de son état
  - Il exposera des méthodes destinées à accéder ou à changer l'état, si nécessaire
  - Traditionnellement, on parle d'**`accesseurs`**
    - Nommé **`getter`** pour une méthode de **`lecture`** de l'attribut
    - Nommé **`setter`** pour une méthode d'**`écriture`** de l'attribut



Autoréférence

- **Un objet peut faire référence à lui-même**
  - En particulier, pour accéder à ses attributs ou s'envoyer lui-même des messages
  - Ou pour se passer en paramètre d'un message ou en retour de méthode
- **Se traduit par une variable spéciale, en général `self` ou `this`**
  - Elle contient à tout moment une référence vers l'objet en cours d'exécution



Fondement n°3
Le polymorphisme

Fondement 3 : Polymorphisme

- **Un fondement qui est une simple conséquence de ce qui précède**
  - Des objets différents peuvent répondre au même message
  - Éventuellement de manière différente
  - Polymorphisme = « plusieurs apparences »
- **Cela diminue considérablement le vocabulaire du logiciel**
  - Il n'est plus nécessaire de trouver un nom unique à chaque fonction
  - Les fonctions AfficherRendezVous(), AfficherAgenda(), AfficherHorloge(), …
  - Deviennent simplement afficher()
  - Un même nom pour la même action

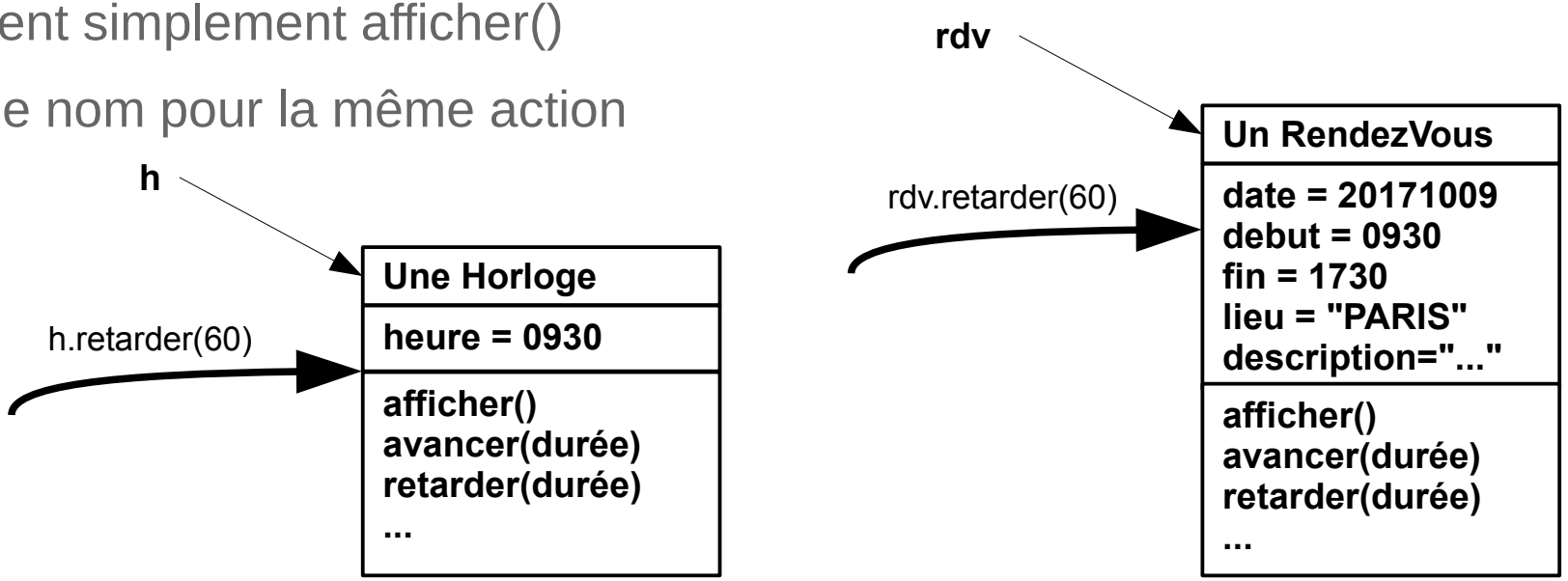


Illustration avec JavaScript

- **Exemple : deux objets `h` et `rdv` (une horloge et un rendez-vous)**
  - Et deux méthodes polymorphes : **`afficher()`** et **`retarder()`**

Objet horloge

Propriété

Méthode

```
var h = {
  heure: 930,
  afficher: function() {
    console.log("Il est "+this.heure)
  },
  retarder: function(duree) {
    this.heure = this.heure + duree
 }
}
```

Messages

```
rdv.retarder(15)
rdv.afficher()
h.retarder(15)
h.afficher()
```

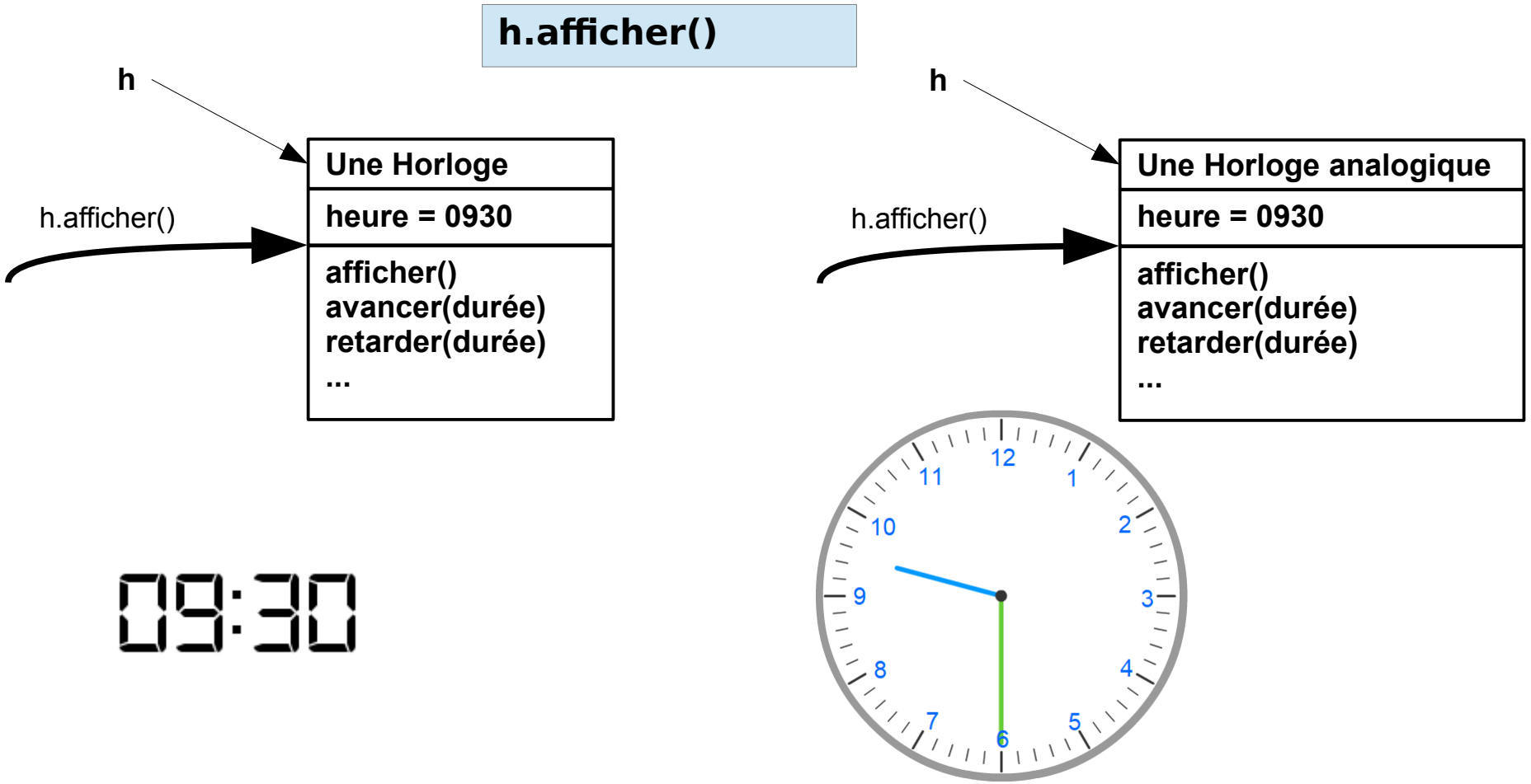
```
FORMATION
 le 20171009
 de 945 à 1745
 à PARIS
Il est 945
```

Objet rendez-vous

```
var rdv = {
  lieu: "PARIS",
  description: "FORMATION",
  date: 20171009,
  debut: 930,
  fin: 1730,
  afficher: function() {
    console.log(this.description)
    console.log(" le " + this.date)
    console.log(" de " + this.debut
                + " à " + this.fin)
    console.log(" à " + this.lieu)
  },
  retarder: function(duree) {
    this.debut = this.debut + duree
    this.fin = this.fin + duree
  }
}
```

Conséquence du polymorphisme

- **La conséquence intéressante : les objets sont interchangeables**
  - On peut remplacer un objet par un autre équivalent ou plus perfectionné
  - Sans modifier le code appelant



Philosophie objet

- **La programmation orientée objet est d'abord conceptuelle**
  - Malheureusement souvent abordée de manière technique dans les langages
  - Et par conséquent souvent mal comprise et mal appliquée
- **C'est avant tout une manière de penser l'organisation du logiciel**
  - On conçoit le logiciel comme un **`écosystème`** constitué d'entités intelligentes, appelés **`objets`**, qui **`collaborent`** en communiquant par **`messages`**
  - Ce n'est ni plus ni moins que l'organisation qu'on retrouve dans une équipe
    - Avec différents rôles spécialisés et les mails, coups de téléphone et discussions pour avancer
- **On peut faire de la POO avec un langage non orienté objet**
  - Il suffit de concevoir en termes d'objets, d'attributs et de méthodes
  - Respecter l'encapsulation, sans accès direct aux attributs
  - Préparer une fonction **`send()`** pour la recherche de méthode
  - Bien sûr, c'est plus simple et efficace si le langage est déjà OO

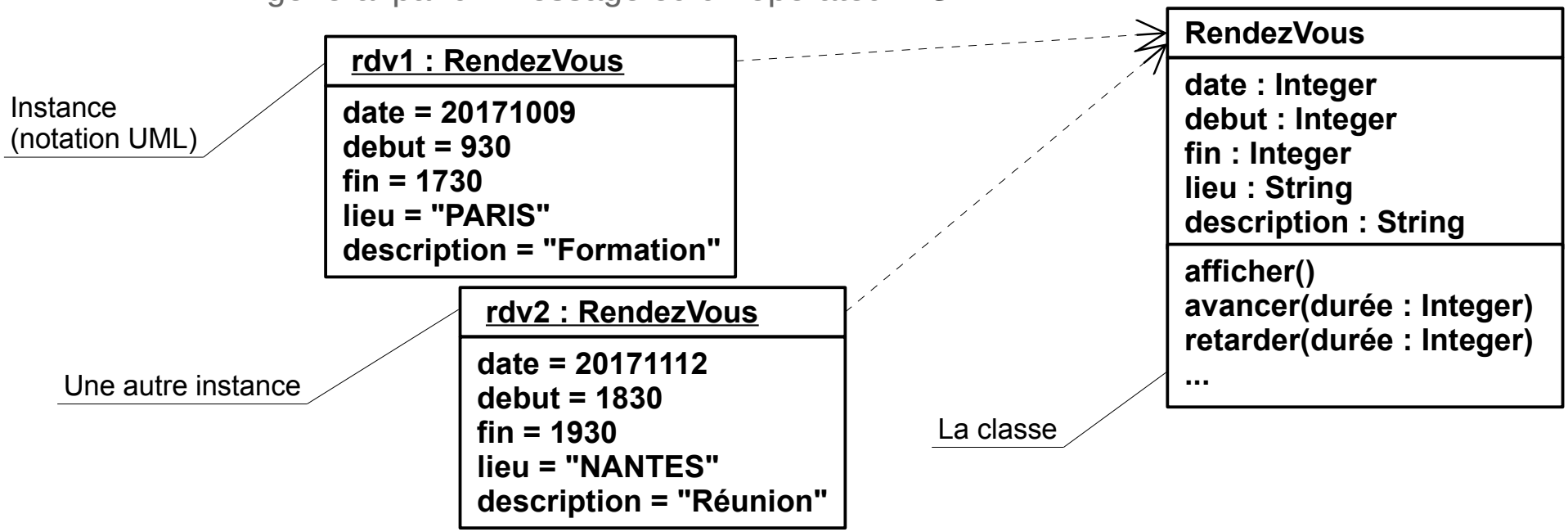
Fondement n°4
Les classes (ou les prototypes)

Code en commun

- **Mais comment fait-on si des objets ont les mêmes méthodes ?**
  - Il serait peu productif de recopier le code dans chaque objet
  - Un agenda devrait contenir de nombreux rendez-vous
  - Chaque rendez-vous devrait pouvoir répondre au message **`afficher()`**
  - Et le code serait identique
- **Deux approches existent pour partager les mêmes méthodes**
  - Approche par **`classe`** : de loin la plus courante (Smalltalk, C++, Java, R, ...)
  - Approche par **`prototype`** : beaucoup plus rare (JavaScript notamment)

La programmation objet à base de classes

- **Notion de `classe`**
  - Pour regrouper la **`définition`** d'objets qui auront les mêmes caractéristiques
  - On définira ainsi la classe des RendezVous avec les propriétés et les méthodes
- **Notion d'`instance`**
  - Les occurrences de la classe, les objets partageant la même définition
  - Elles sont créées à partir de la classe, on parle d'**`instanciation`** de la classe
    - En général par un message ou un opérateur **`new`**



Classe et recherche de méthode

- **C'est maintenant la classe qui contient les méthodes**
  - Les instances gardent, en interne, un lien vers leur classe
  - La recherche de méthode est donc techniquement un peu différente
  - Lorsqu'une instance reçoit un message, elle cherche la méthode dans la classe
  - Et l'exécute ensuite pour son compte (avec les valeurs de ses propres attributs)

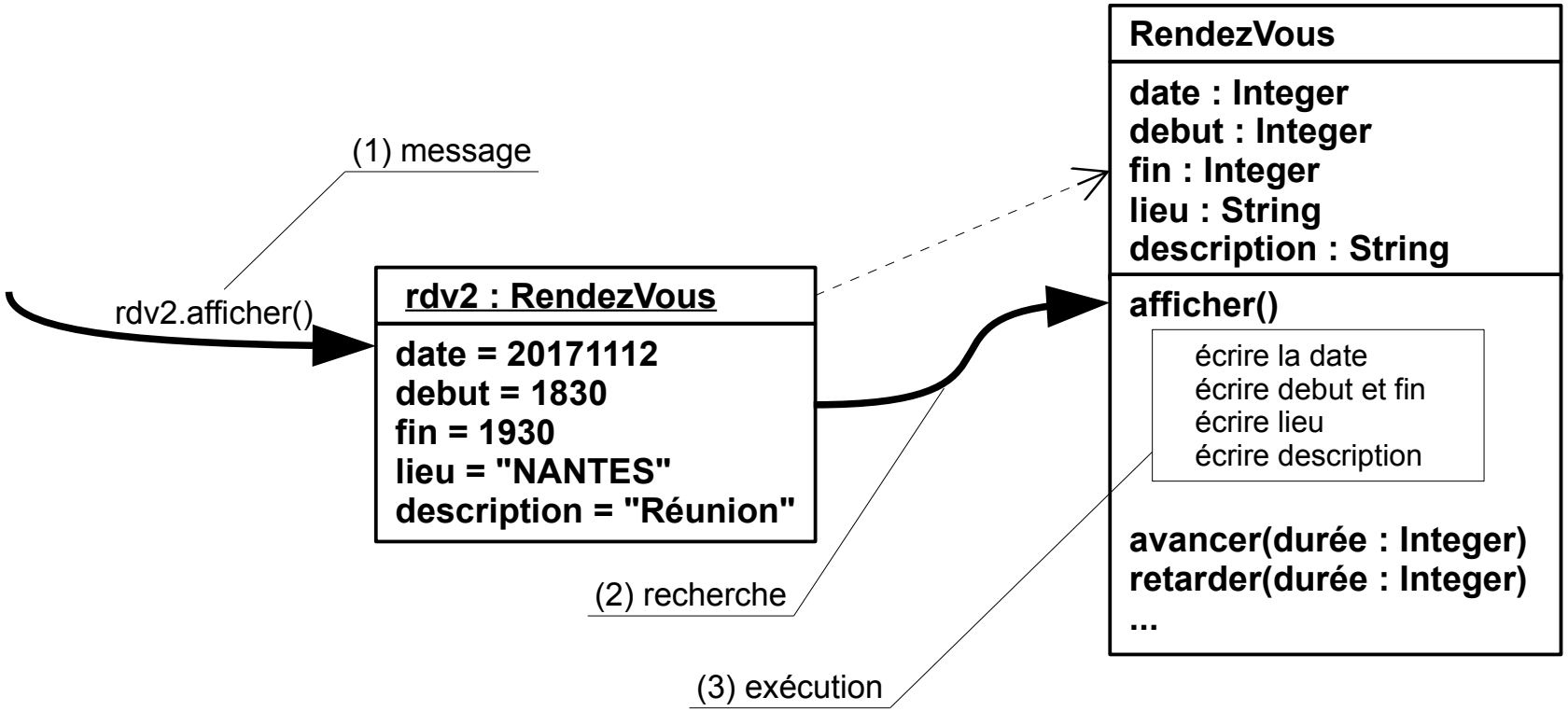


Illustration avec Java

```
public class RendezVous {
   int date = 20170101;
   int debut = 930;
   int fin = 1730;
   String lieu = "Inconnu";
   String description = "Formation";

   public void afficher() {
      System.out.println("Le " + date);
      System.out.println("de " + debut + " à " + fin);
      System.out.println("à " + lieu);
      System.out.println(description);
   }
   public void retarder(int duree) {
      debut = debut + duree;
      fin = fin + duree;
   }
   public static void main(String[] args) {
      RendezVous rdv1 = new RendezVous();
      RendezVous rdv2 = new RendezVous();
      rdv1.afficher();
      rdv1.retarder(15);
      rdv1.afficher();
      rdv2.afficher();
   }
}
```

Déclaration de la classe

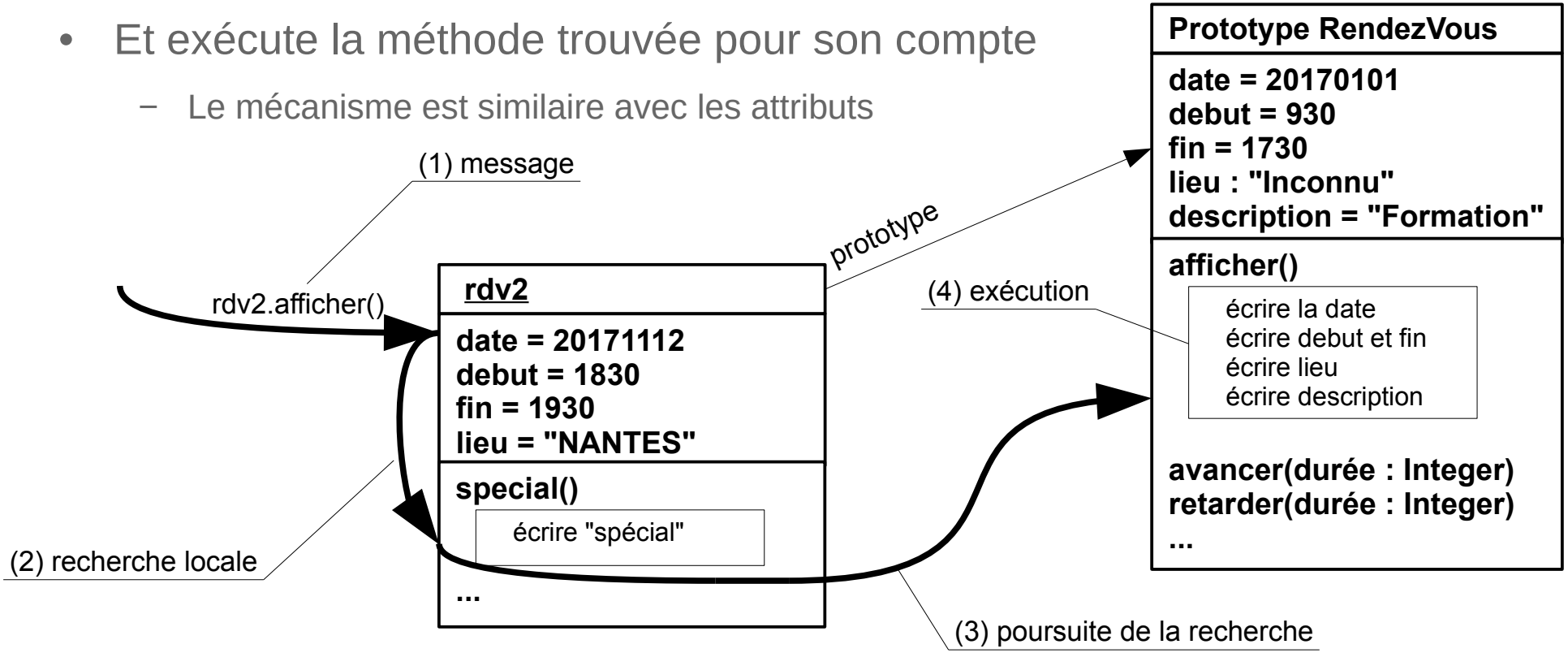
Attributs et valeurs par défaut

Méthodes

Création de deux instances

```
Le 20170101
de 930 à 1730
à Inconnu
Formation
Le 20170101
de 945 à 1745
à Inconnu
Formation
Le 20170101
de 930 à 1730
à Inconnu
Formation
```

- **Dans un système à base de prototypes, il n'y a pas de classe**
  - Chaque objet contient ses propres méthodes
  - Il est créé par clonage d'un autre objet, son **`prototype`**, et garde un lien vers lui
  - Lorsqu'il reçoit un message, il commence par rechercher localement la méthode
  - S'il ne trouve pas, il recherche dans son prototype
  - Et exécute la méthode trouvée pour son compte
    - Le mécanisme est similaire avec les attributs



Illustration

- **JavaScript est un célèbre langage à base de prototype**
  - Il peut facilement être testé dans un navigateur moderne (Ctrl+Maj+i)
  - **Remarque :** ce code est purement illustratif (pas un exemple à suivre)

```
var ProtoRendezVous = {
  date : 20170101,
  debut : 930,
  fin : 1730,
  lieu : 'Inconnu',
  description : 'Formation',

  afficher : function () {
    console.log('Le ' + this.date)
    console.log('de ' + this.debut + ' à ' + this.fin)
    console.log('à ' + this.lieu)
    console.log(this.description)
  },

  retarder : function (duree) {
    this.debut = this.debut + duree
    this.fin = this.fin + duree
  }
}
```

Création d'un objet prototype

Attributs et valeurs par défaut

Méthodes

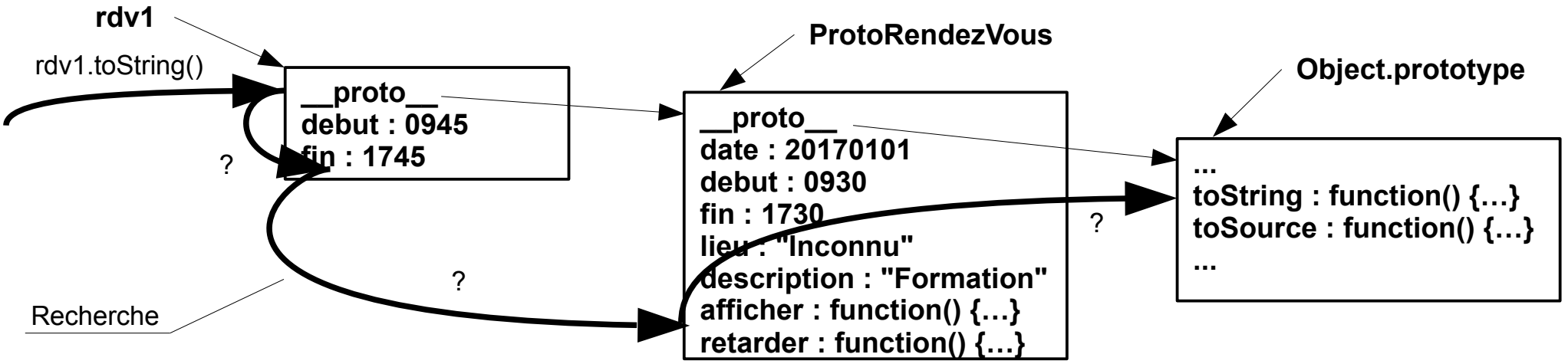
Création de deux instances

```
var rdv1 = Object.create(ProtoRendezVous)
var rdv2 = Object.create(ProtoRendezVous)
rdv1.afficher();
rdv1.retarder(15);
rdv1.afficher();
rdv2.afficher()
```

Fondement n°5
L'héritage

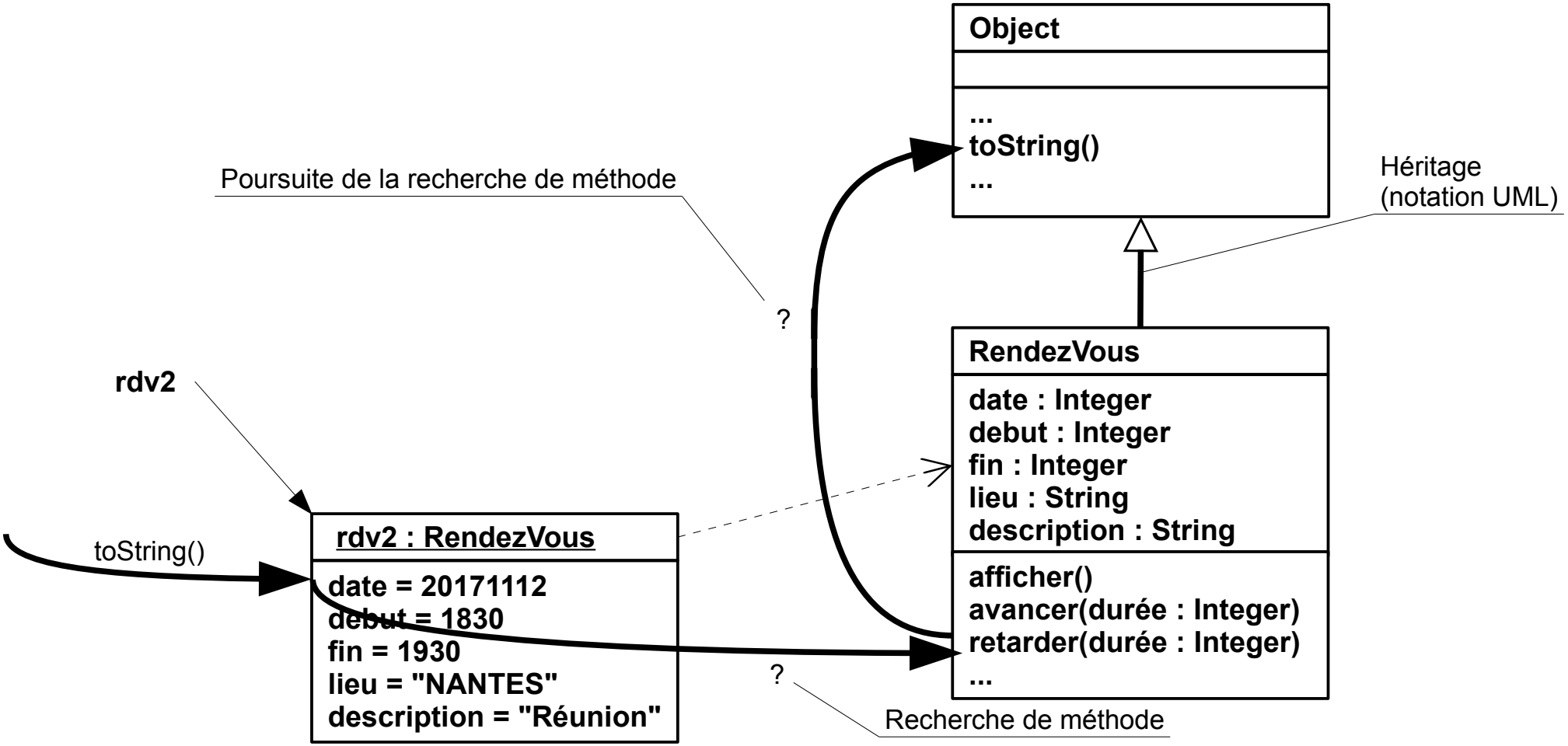
Héritage avec les prototypes

- **On dit que rdv1 `hérite` des méthodes de son prototype**
  - Puisqu'il récupère automatiquement des méthodes qu'il ne contient pas
- **Mais rdv1 hérite d'autres méthodes non écrites dans le prototype**
  - Par exemple `rdv1.toString()` ou `rdv1.toSource()`
  - D'où peuvent bien provenir ces méthodes ?
- **L'attribut `__proto__` nous donne la réponse**
  - Notre ProtoRendezVous a lui-même un prototype, celui commun aux objets
  - La méthode `toString()` est héritée indirectement de ce dernier prototype



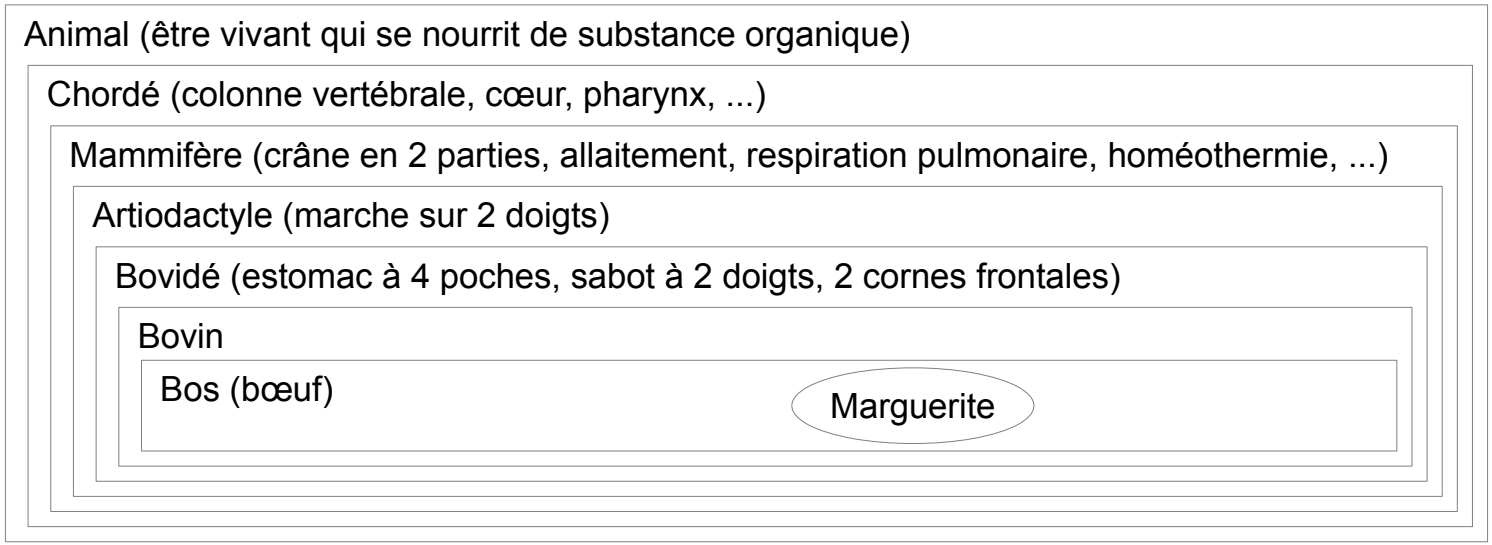
Héritage avec les classes

- **Dans les langages de classes, l'héritage est porté par les classes**
  - Une classe peut référencer une **`superclasse`**
  - La recherche de méthode est alors prolongée dans la chaîne des superclasses



Héritage

- **L'idée de l'héritage est de décrire de nouvelles classes par delta**
  - Les **sous-classes** héritent des attributs et des méthodes
    - Comme dans la classification des animaux où les espèces sont **classées** selon des **critères**
    - Chaque sous-branche ou sous-boite **hérite** des caractéristiques de la branche/boite **parente**
  - Ici encore, l'héritage est d'abord conceptuel, c'est une manière d'organiser
    - Les langages objets apportent leurs spécificités et contraintes (héritage multiple ou simple, …)
    - En général, l'héritage simple est retenu, la **racine de l'arbre** s'appelle souvent **Object**



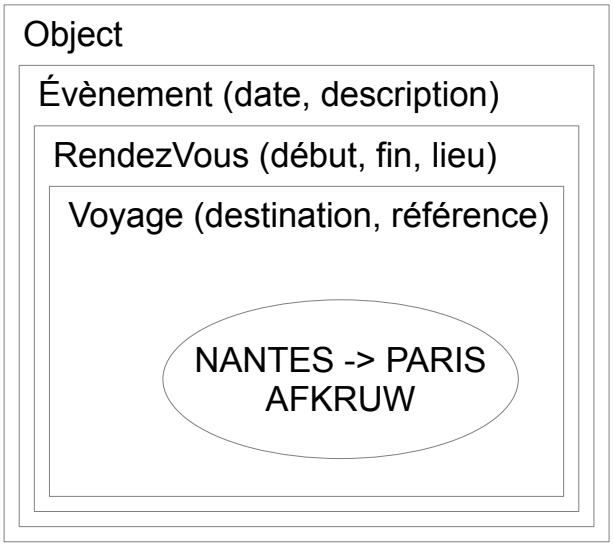


Illustration avec Java

- **Création d'une classe Voyage héritant de la classe RendezVous**
  - Un Voyage serait une sorte de RendezVous dans l'agenda (date, horaires, ...)
  - On y ajouterait la notion de « destination » et de « référence dossier »

```
public class Voyage extends RendezVous {
   private String destination = "Inconnue";
   private String refDossier = "??????";

   public String getDestination() {
      return destination;
   }

   public void setDestination(String destination) {
      this.destination = destination;
   }
   ...

   public static void main(String[] args) {
      Voyage v = new Voyage();
      v.afficher();
      v.setDestination("PARIS");
   }
}
```

Déclaration de l'héritage

Attributs supplémentaires (en plus de ceux hérités)

Getter de l'attribut destination

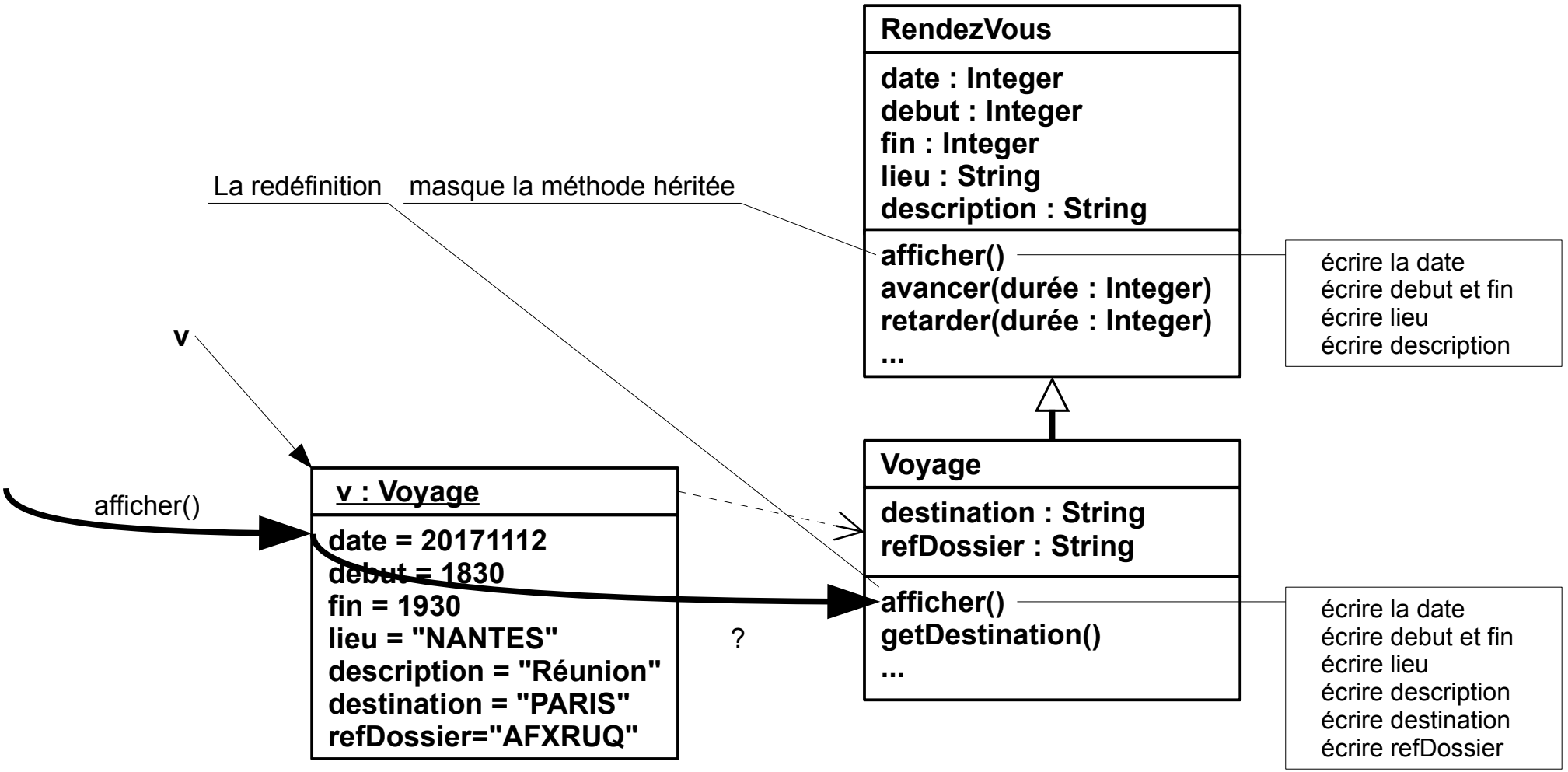
Setter de l'attribut destination

Méthode afficher() héritée

```
Le 20170101
de 930 à 1730
à Inconnu
Formation
```

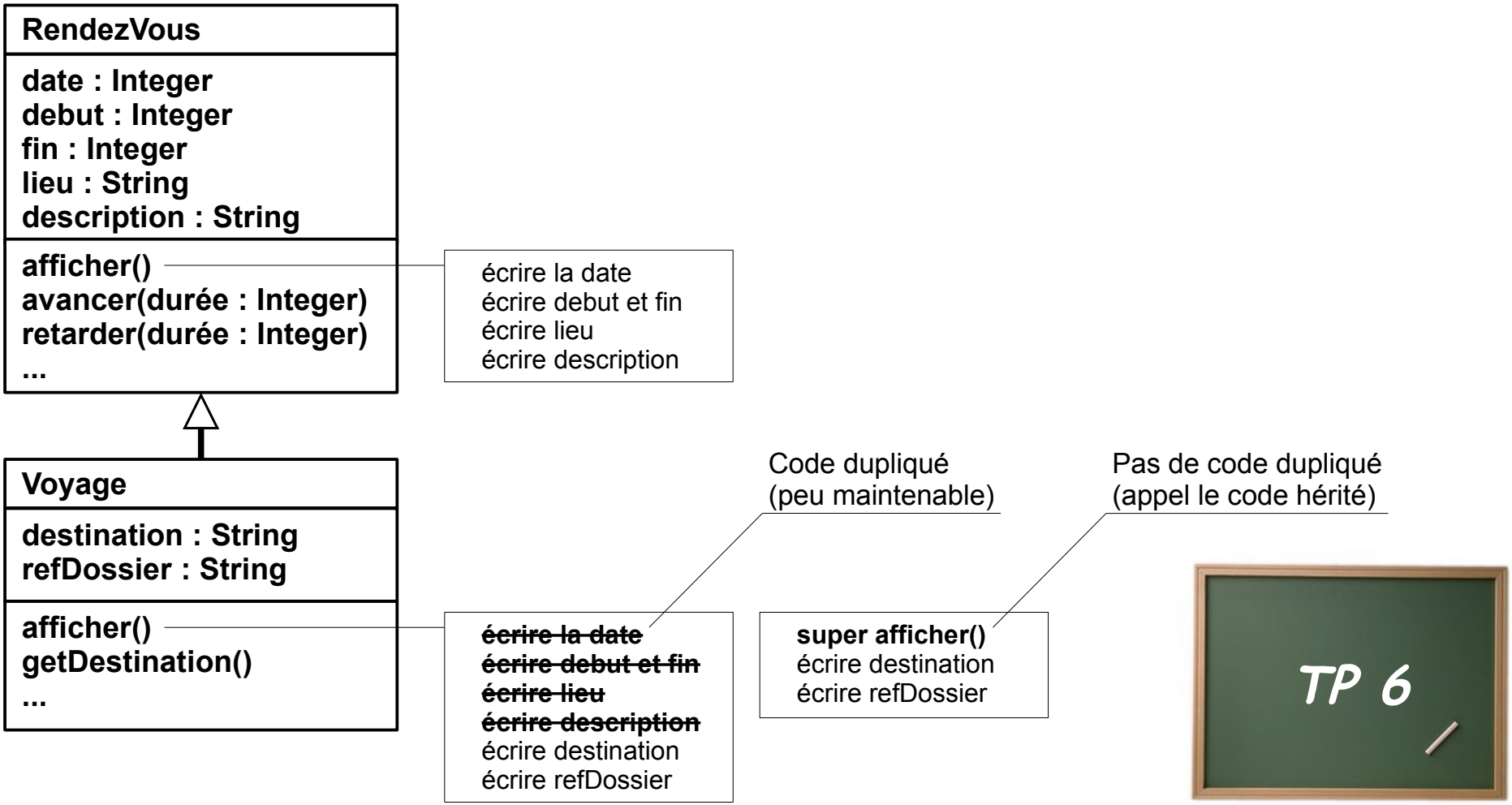
Héritage et redéfinition

- **Mais comment afficher correctement un Voyage ?**
  - Il manque la destination et la référence dossier dans l'affichage !
- **La recherche de méthode permet la `redéfinition` de méthode**



Héritage et « super »

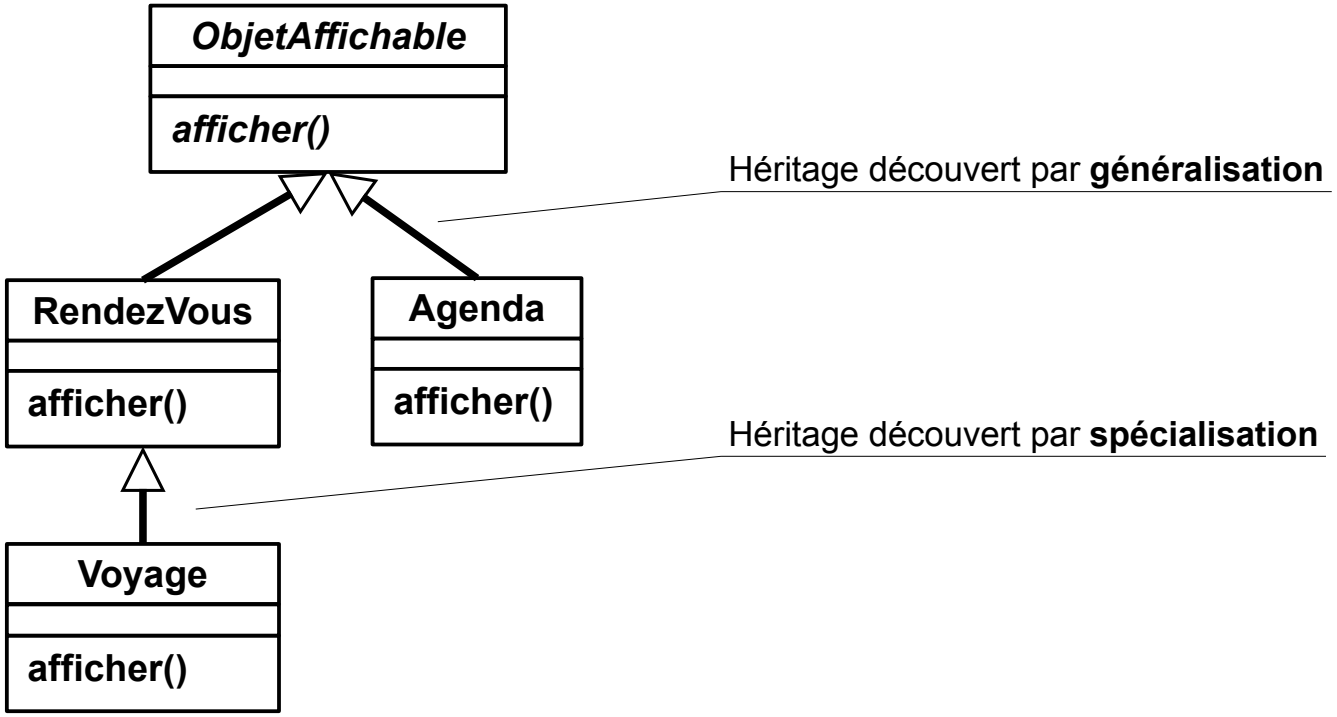
- **Mais comment éviter la recopie de code dans cette redéfinition ?**
  - Si la superclasse change la méthode **`afficher()`**, la copie ne sera pas à jour !
  - Les langages objets introduisent une pseudo variable **`super`** pour cette situation



Fondement n°6
L'abstraction

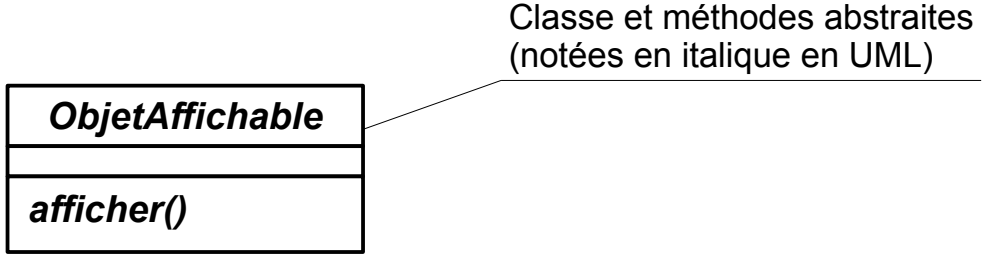
Spécialisation et généralisation

- **L'héritage est une relation entre deux classes**
  - Découverte par **`spécialisation`** : j'identifie un cas spécial
    - Exemple : j'identifie un cas spécial de **`RendezVous`** : le **`Voyage`**
  - Découverte par **`généralisation`** : j'identifie un cas général
    - Exemple : RendezVous et Agenda et proposent **`afficher()`**
    - Je peux construire le concept plus général d'**`ObjectAffichable`**



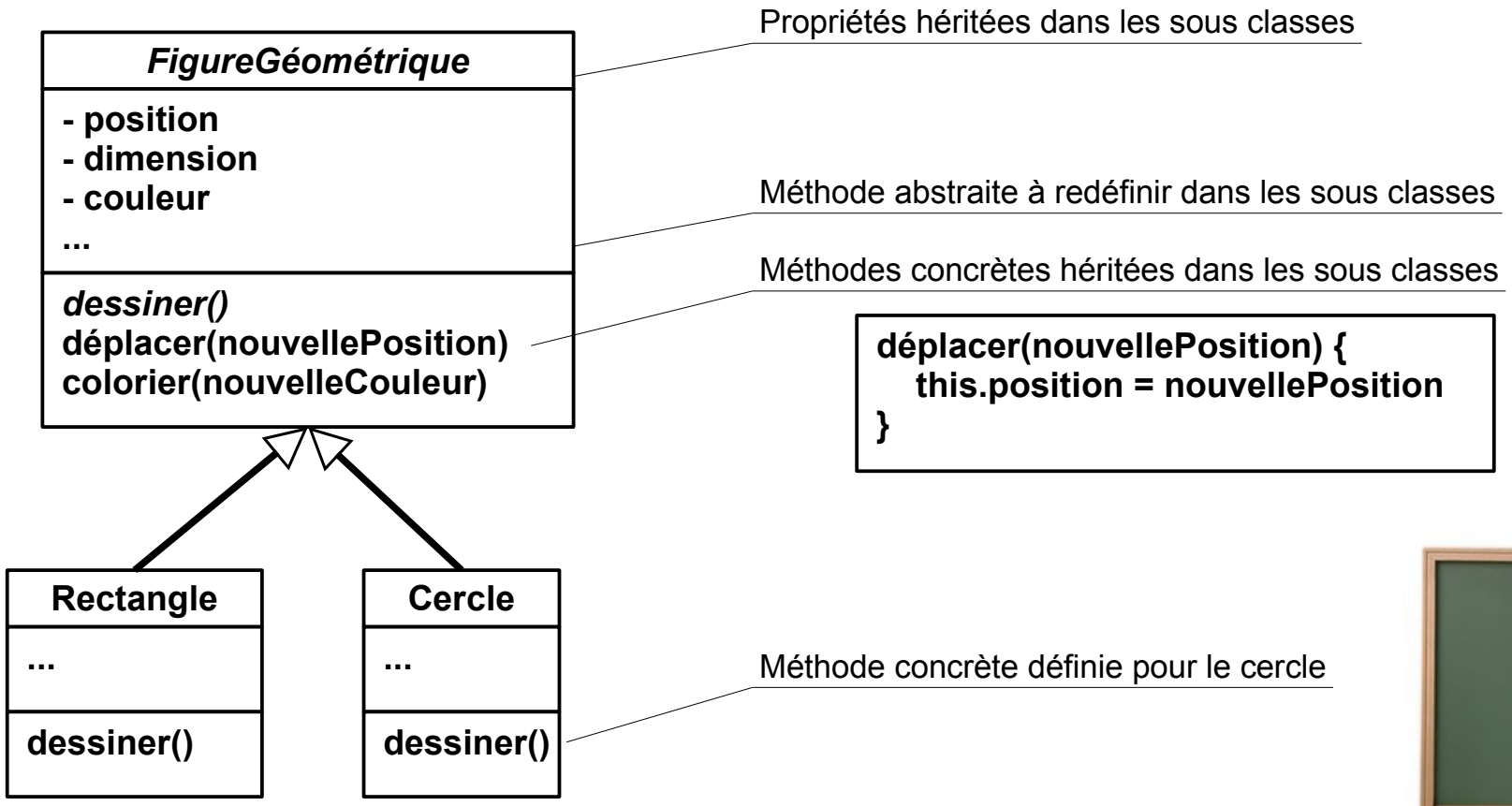
Abstraction

- **La découverte par généralisation conduit souvent à l'abstraction**
  - J'identifie un cas général permettant de factoriser (mettre en commun)
    - Exemple : je vois un **`chêne`**, avec son tronc, ses racines, ses branches, ses feuilles
    - Je vois aussi un **`Peuplier`**, lui aussi avec son tronc, ses racines, ses branches, ses feuilles
    - Et je vois encore un **`Marronnier`**, un Sapin et un **`Saule`**
    - Et je désigne cela par un terme général : ce sont tous des sortes d'**`Arbre`**
  - Le cas général est souvent **`abstrait`**, il n'en existe pas d'instance
    - La notion d'arbre est purement abstraite (et humaine), liée à la classification du vivant
    - On ne peut pas montrer un arbre, ce sera toujours un Marronnier, un Peuplier, un Chêne
  - Certaines méthodes sont aussi **`abstraites`**
    - On ne sait pas définir le comportement par défaut (ex : faire des branches)
    - Comment afficher un **`ObjectAffichable`** ? On ne sait pas, on sait qu'on peut l'**`afficher()`**
    - On exprime l'abstraction avec UML en utilisant une écriture italique



Abstrait ou concret ?

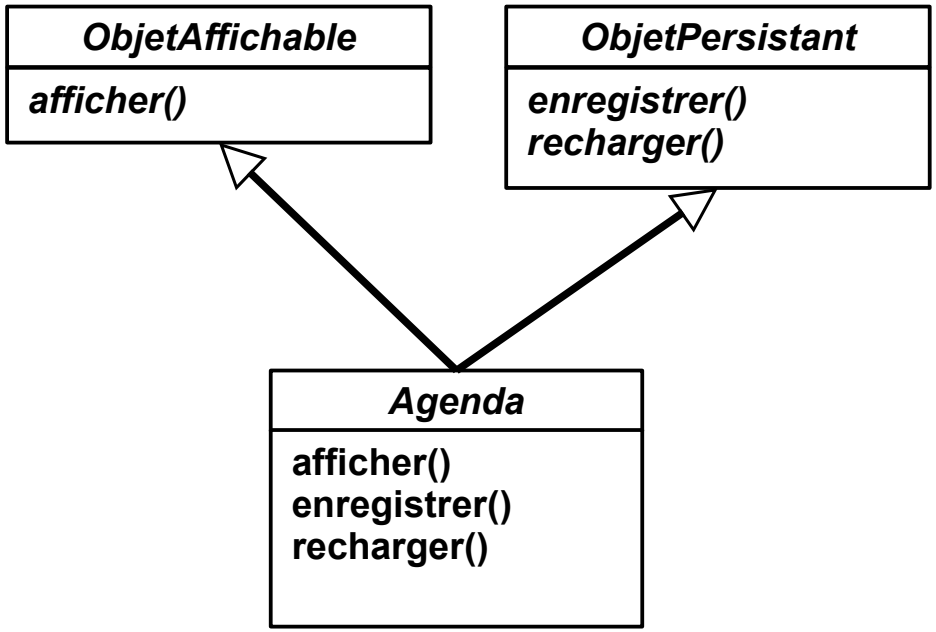
- **Les classes abstraites peuvent contenir des méthodes concrètes**
  - Ainsi que des propriétés
  - Comment afficher un **`ObjetAffichable`** ?
    - On ne sait pas, on sait qu'on peut l'**`afficher()`**



Notions avancées

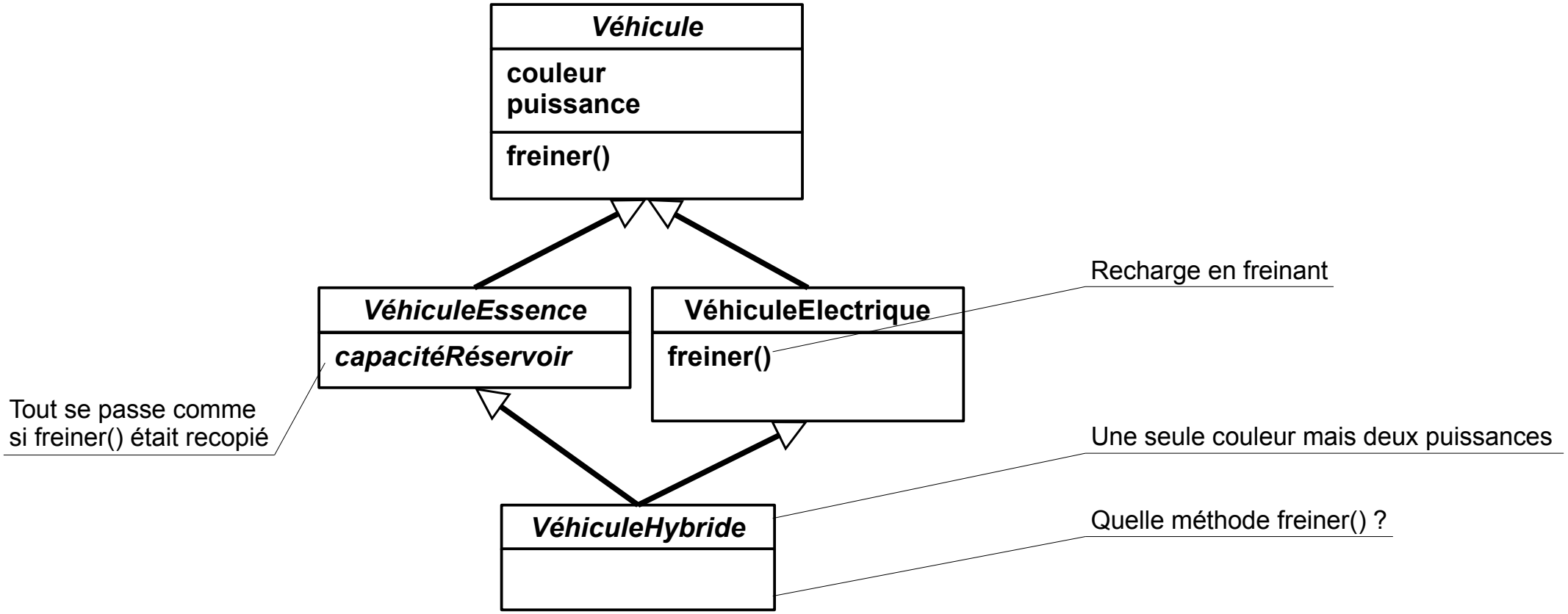
Héritage multiple

- **Ajoutons un concept de persistance**
  - Un **`ObjetPersistant`** peut s'**`enregistrer()`** et se **`recharger()`**
  - On veut exprimer que notre **`Agenda`** est à la fois **`affichable`** et **`persistant`**
- **L'héritage multiple consiste à indiquer plusieurs superclasses**
  - Semble intéressant et est supporté par certains langages



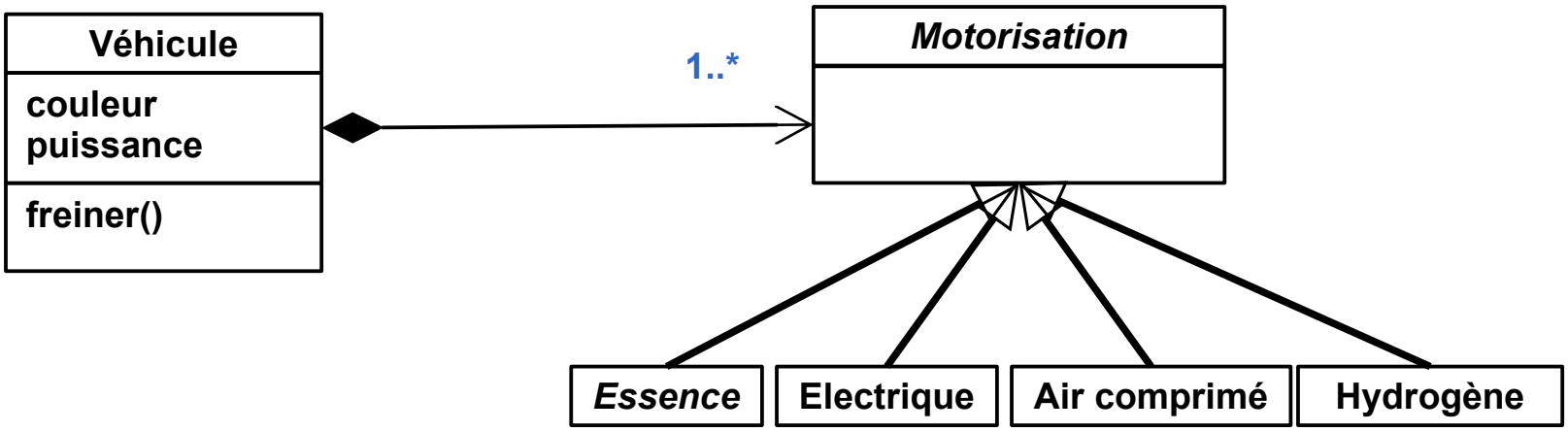
Problèmes de l'héritage multiple

- **L'héritage multiple apporte plusieurs problèmes**
  - On ne sait pas choisir automatiquement si l'héritage des propriétés est répété
  - On ne sait pas choisir automatiquement de quelle méthode hériter
    - L'héritage n'est qu'une économie de recopie, on pourrait recopier
  - La plupart des langages objets ont abandonné l'idée de l'héritage multiple



Préférer la composition

- **La composition (l'assemblage) offre une meilleure réutilisation**
  - Se méfier des noms composés
  - Oblige à écrire plus de code mais offre plus d'opportunité de réutilisation
- **Exemple**
  - Un véhicule comporte une motorisation ou plusieurs
  - Il existe plusieurs sorte de motorisation

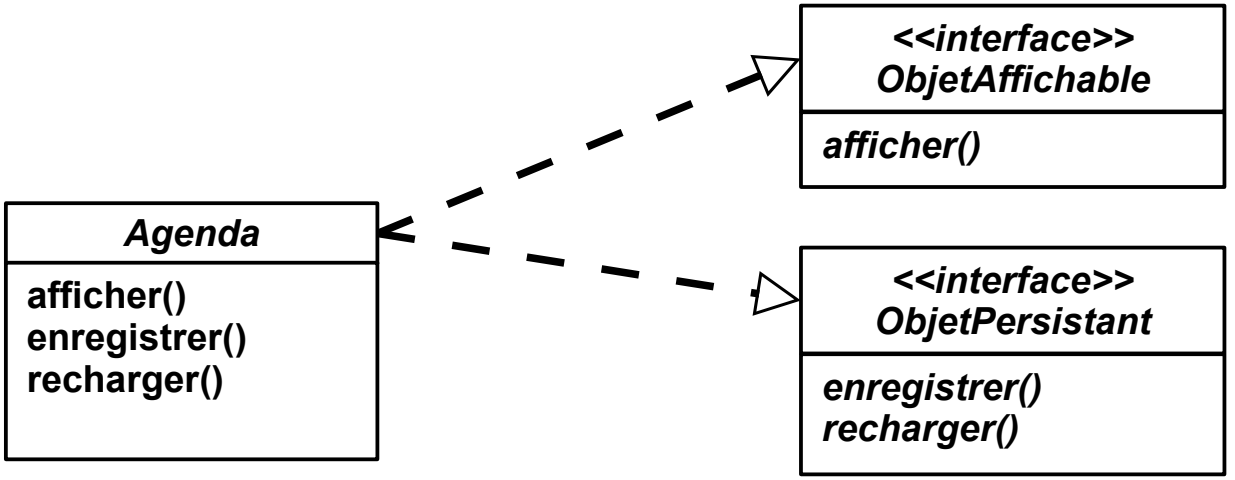


Classe abstraite = type abstrait

- **Une classe représente un `type` d'objet**
  - Sert à typer les variables dans un langage typé comme Java par exemple
- **Une classe abstraite représente un `type abstrait`**
  - Permet d'accepter n'importe quel sous type
- **L'héritage est un lien trop fort et trop contraignant**
  - Problèmes précédents avec l'héritage multiple
- **On a besoin d'exprimer un type purement abstrait : l'`interface`**
  - Permet de définir un ensemble de méthodes abstraites sans imposer d'héritage
- **Grâce à ce concept, on peut définir des classes `multi-types`**
  - Et ainsi partiellement résoudre le problème de l'héritage multiple

Interface

- **Une interface correspond à un type purement abstrait**
  - Identifie un ensemble d'opérations abstraites : un contrat
  - Permet de concevoir et même d'hériter de manière multiple
  - Une classe peut réaliser (implémenter) une ou plusieurs interfaces
    - Notation flèche d'héritage en tirets avec UML
  - Et cela indépendamment de son héritage
- **Exemple**
  - Agenda réalise le contrat des objets affichables et des objets persistants



Conclusion

Conclusion

- **Un aperçu des concepts fondamentaux de la POO**
  - Pas toujours aussi bien compris par les développeurs censés les maîtriser
- **Une présentation théorique**
  - Ces concepts se retrouvent dans les différents langages objets
  - Avec parfois d'autres concepts
- **Difficile à appréhender en une journée**
  - Il faut des mois de pratique assidue pour maîtriser ces concepts
  - C'est un changement de paradigme, une autre manière de penser
- **Persévérer**
  - Reprendre tranquillement les pages du cours et les exercices
  - Surtout après avoir appris le langage cible