0-M02 – Introduction à la programmation
Introduction à l'algorithmique

Cours n°2
Programmation structurée
Introduction à la conception orientée objet

Camille Coti
camille.coti@iutv.univ-paris13.fr

IUT de Villetaneuse, département R&T

2011 – 2012

Décomposition d'un problème en sous-problèmes

"... diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour mieux les résoudre."

Discours de la méthode (1637)
René Descartes

Fonctions

Définition

Une **fonction** définit une action effectuée sur un ensemble de paramètres qui produit un résultat retourné comme valeur de sortie de la fonction.
Elle fournit une **abstraction** pour cette action vis-à-vis des algorithmes qui l'appellent.

Analogie avec les fonctions mathématiques : $f : x \mapsto f(x)$

- On définit la fonction une fois
- On l'appelle autant de fois qu'on veut

Avantages : factorisation de code, abstraction pour la conception de l'algorithme... Permet de se concentrer sur l'algorithme lui-même plutôt que sur les détails.

Paramètres et résultat

On donne les paramètres et leur type

- On déclare les paramètres entre parenthèses
- On les utilise dans la fonction en utilisant **ce nom** : ce sont des variables
- Il existe des paramètres
  - D'entrée : utilisés dans la fonction
  - De sortie : modifiés dans la fonction
  - Ou les deux : utilisés et modifiés dans la fonction

On déclare le type retourné

- Déclaration de la fonction suivie par **: type**
- On sort de la fonction avec le mot clé **retourner** et la valeur retournée

```
1 début fonction carre( nombre: Entier ): Entier
2 |   resultat : Entier
3 |   resultat ⟵ nombre × nombre
4 |   retourner resultat
5 fin fonction
```

Appel de fonction

On appelle la fonction à partir d'un autre algorithme

- On lui passe en paramètres des variables de l'algorithme appelant
- La valeur retournée est mise dans une variable de l'algorithme appelant

```
1 début programme
2 |   nombreDepart : Entier
3 |   calcul : Entier
4 |   calcul ⟵ carre ( nombreDepart )
5 fin programme
```

```
1 début fonction carre( nombre: Entier ): Entier
2 |   resultat : Entier
3 |   resultat ⟵ nombre × nombre
4 |   retourner resultat
5 fin fonction
```

La fonction fournit donc une **abstraction** de l'action qu'elle réalise paramétrée par les variables passées lors de l'appel.

Procédures

Une procédure effectue une **action**. Elle ne retourne rien.

- Affichage : pas de résultat de calcul à récupérer
- Calcul sur des **variables de sortie**

On sort de la procédure

- À la fin du bloc d'instructions principal
- Ou quand on rencontre le mot-clé **Retour**

Procédures (exemple)

```
1 début programme
2 |   puissance : Entier
3 |   puissance ( puissance )
4 |   afficher ( puissance )
5 fin programme
```

```
1 début procédure puissance( petitepuissance: Entier Sortie )
2 |   petitepuissance ⟵ 1
3 |   tant que Vrai faire
4 |   |   petitepuissance ⟵ petitepuissance × 2
5 |   |   if petitepuissance > 200 then
6 |   |   |   retourner
7 |   |   endif
8 |   fintq
9 fin procédure
```

Visibilité des variables

Les variables déclarées dans la fonction

- ne sont visibles **que** dans la fonction

Les variables déclarées dans le programme appelant

- ne sont visibles **que** dans le programme appelant
- ne sont **pas** visibles dans la fonction

Les variables passées en paramètre

- sont appelées par leur nom dans le programme appelant dans l'appel de la fonction
- sont appelées par le nom utilisé pour les déclarer dans la fonction elle-même

Visibilité des variables

Les variables sont visibles uniquement dans la fonction, la procédure ou le programme dans lequel elles ont été déclarées, et dans aucune des fonctions ou procédures appelées ou qui l'appellent.

Approche descendante

Les fonctions et les procédures fournissent une **abstraction sur les actions réalisées par le programme**

- Possibilité de les utiliser pour se concentrer sur la structure du programme plutôt que sur la résolution des sous-problèmes

On décompose le problème en **sous-problèmes**

- On fait dans un premier temps l'hypothèse qu'ils sont résolus
- On s'en occupe plus tard

La conception de l'algorithme dans sa globalité est ainsi **plus simple**

- Décomposer pour mieux maîtriser

On retarde le plus possible le moment d'effectuer les calculs

Approche descendante : exemple

Théorème de Pythagore : *"le carré de l'hypoténuse d'un triangle rectangle est égal à la somme des carrés des longueurs des deux autres côtés"*.

```
1  début fonction hypo( cote1: Entier, cote2: Entier ): Entier
2      /* variables internes */
3      carre1 : Entier
4      carre2 : Entier
5      hypotenusecarre : Entier
6      hypotenuse : Entier
7      /* on calcule la somme des carrés des côtés */
8      carre1 ⟵ carré( cote1 )
9      carre2 ⟵ carré( cote2 )
10     hypotenusecarre ⟵ carre1 + carre2
11     /* on prend la racine carrée de la somme et on retourne le
       résultat */
12     hypotenuse ⟵ racine( hypotenusecarre )
13     retourner hypotenuse
14 fin fonction
```

Approche descendante : exemple (suite)

```
1 début fonction carré( nombre: Entier ): Entier
2 |   moncarre : Entier
3 |   moncarre ⟵ nombre × nombre
4 |   retourner moncarre
5 fin fonction
```

```
1 début fonction racine( nombre: Entier ): Entier
2 |   maracine : reel
3 |   maracine ⟵ √nombre
4 |   retourner maracine
5 fin fonction
```

Récursivité

Une fonction ou une procédure peut **s'appeler elle-même**

- Elle est alors partiellement définie à partir d'elle-même

```
1 début fonction calcul( var: Entier ): Entier
2 |   si var == 1 alors
3 |   |   retourner var
4 |   sinon
5 |   |   retourner var×calcul( var − 1 )
6 |   finsi
7 fin fonction
```

- Correspond bien aux relations de récurrence
- Attention au point d'arrêt !

Intérêt de la COO

La conception orientée objet fournit une abstraction sur les données manipulées

- Briques de base du programme
- Le programme manipule ces "briques" qui interagissent entre elles

On **rassemble** les données relatives à l'état d'un élément du système et les actions dans une structure de données particulière : **une classe**.

Conception d'un système à partir de l'analyse du problème :

- Identification des objets
- Définition des actions réalisées sur les objets et leurs interactions
- Identification des données représentatives de l'état des objets

Définition d'une classe

Une classe est constituée de deux catégories d'éléments :

- Des attributs
- Des méthodes

Attributs

Les attributs sont les données relatives à l'objet d'une classe et représentatives de son état. Ce sont des variables conservées dans la mémoire de l'objet manipulé.

Méthodes

Les méthodes décrivent le comportement des objets et les actions que l'on peut effectuer dessus. Ce sont des fonctions ou des procédures.

Variables

Objet

Un **objet** est une **instance** d'une classe. La classe est la définition de la structure de données, l'objet est la représentation en mémoire de cette classe qui est manipulée par le programme.

Attention à la distinction :

- Une classe est la description des objets
- Un objet est une instance d'une classe. On peut avoir plusieurs objets d'une même classe.

Portée des champs d'un objet

Les attributs et les méthodes peuvent être :

- **Privés** : accessibles uniquement depuis l'objet
- **Publics** : accessible depuis n'importe quel objet

Intérêt : ne pas se tromper !

- Pas de modification directe des attributs par un autre objet

Exemple : la classe Moto

Comment caractériser une moto ?

- Une moto a des roues (utilisation d'une classe Roue), un guidon, une quantité d'essence dans le réservoir, une consommation
- Elle peut être garée ou rouler à une certaine vitesse

On a besoin d'une classe Roue. Comment la caractériser ?

- Une roue a un diamètre et elle est gonflée à un certain niveau

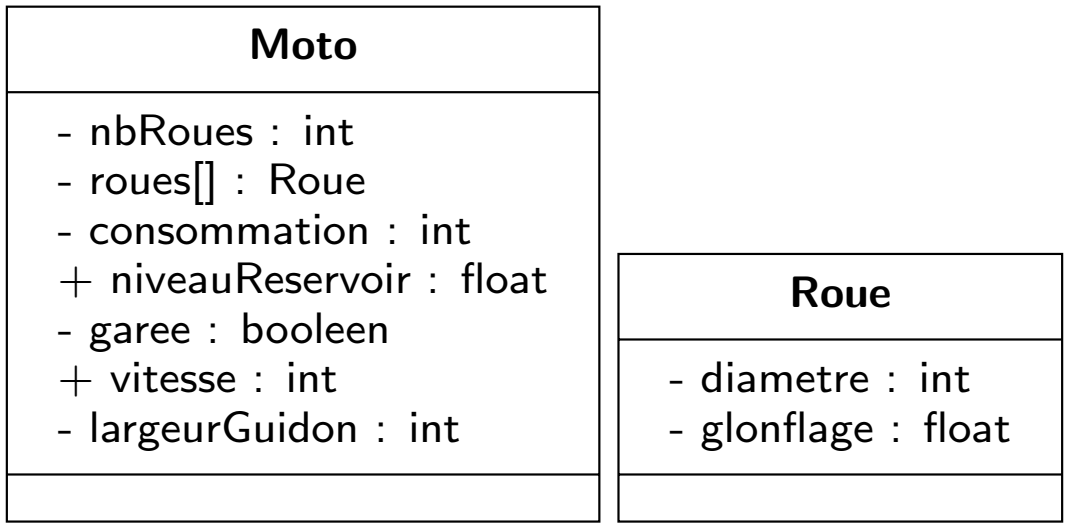
Exemple : la classe Moto

Comment caractériser une moto ?

- Une moto a des roues (utilisation d'une classe Roue), un guidon, une quantité d'essence dans le réservoir, une consommation
- Elle peut être garée ou rouler à une certaine vitesse

On a besoin d'une classe Roue. Comment la caractériser ?

- Une roue a un diamètre et elle est gonflée à un certain niveau



Exemple : la classe Moto (suite)

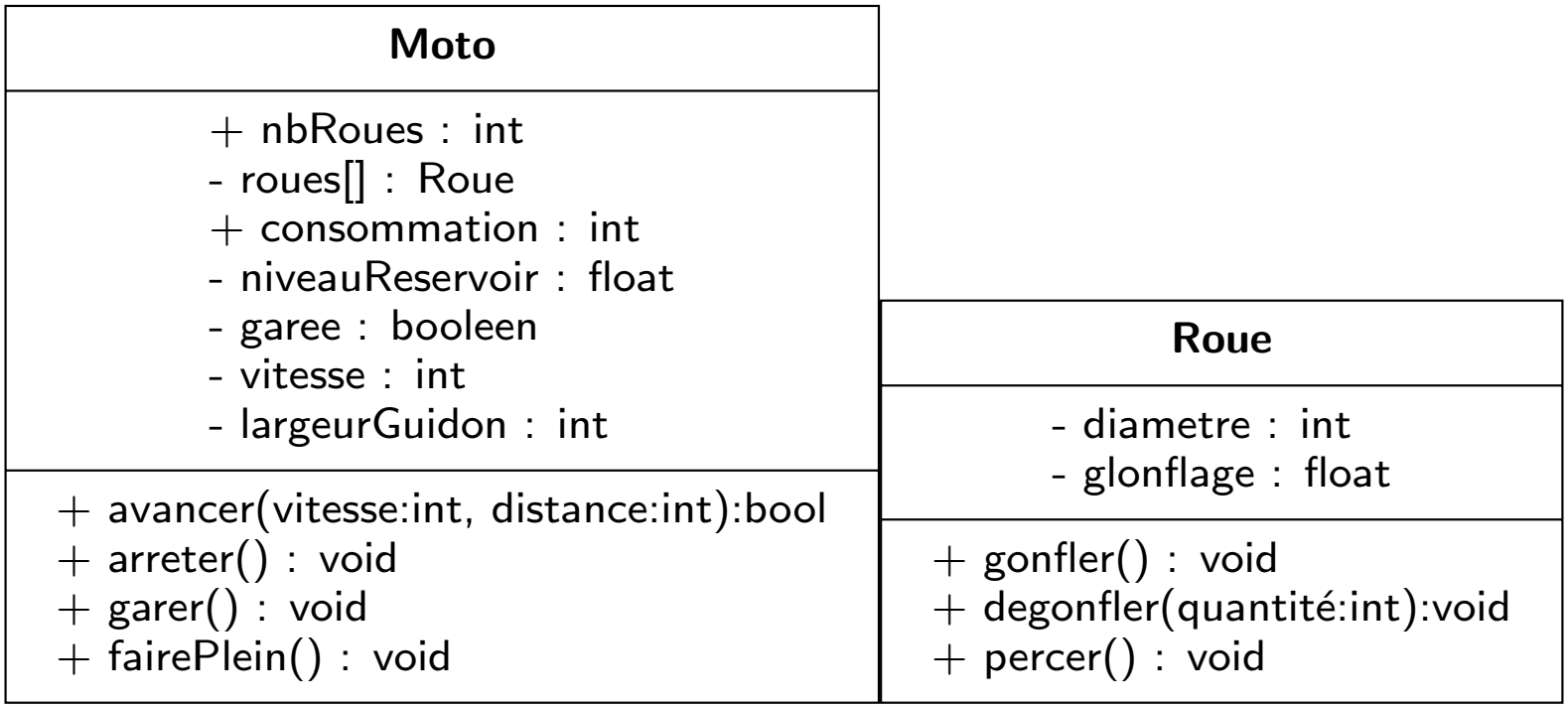
Quelles actions peut-on effectuer sur une moto ?

- Avancer à une certaine vitesse sur une certaine distance : on a une méthode `bool avancer( int vitesse, int distance )`
- S'arrêter : on a une méthode `void arreter()`
- Se garer : on a une méthode `void garer()` qui arrête la moto si besoin et qui la gare
- Remplir le réservoir d'essence : `void fairePlein()`

Quelles actions peut-on effectuer sur une roue ?

- La gonfler complètement : `void gonfler()`
- La dégonfler partiellement : `void degonfler( int quantité )`
- La percer (dégonfler complètement) : `void percer()`

Exemple : la classe Moto (suite)



Portée des attributs et des méthodes :

- Privé : noté -
- Public : noté +

Des classes supplémentaires

En plus de nos classes Moto et Roue, on défini deux autres classes Voiture et Camion :

Voiture
+ nbRoues : int - roues[] : Roue + consommation : int - niveauReservoir : float - garee : booleen - vitesse : int - diametreVolant : int - capaCoffre : int - remplissageCoffre : float
+ avancer(vitesse:int,distance:int): bool + arreter() : void + garer() : void + fairePlein() : void + remplirCoffre(volume : float) : bool + viderCoffre( ) : void

Camion
+ nbRoues : int - roues[] : Roue + consommation : int - niveauReservoir : float - garee : booleen - vitesse : int - diametreVolant : int - nbRemorques : int - remorques[] : Remorque
+ avancer(vitesse:int,distance:int): bool + arreter() : void + garer() : void + fairePlein() : void + ajouterRemorque() : void

Héritage

On constate des attributs et des méthodes communs entre Moto, Camion et Voiture :

- Attributs :
  - nbRoues : int
  - roues[] : Roue
  - consommation : int
  - niveauReservoir : float
  - garee : booleen
  - vitesse : int
- Méthodes :
  - avancer(vitesse : int, distance : int) : bool
  - arreter() : void
  - garer() : void
  - fairePlein() : void

Héritage

On va donc les **rassembler** dans une classe Vehicule :

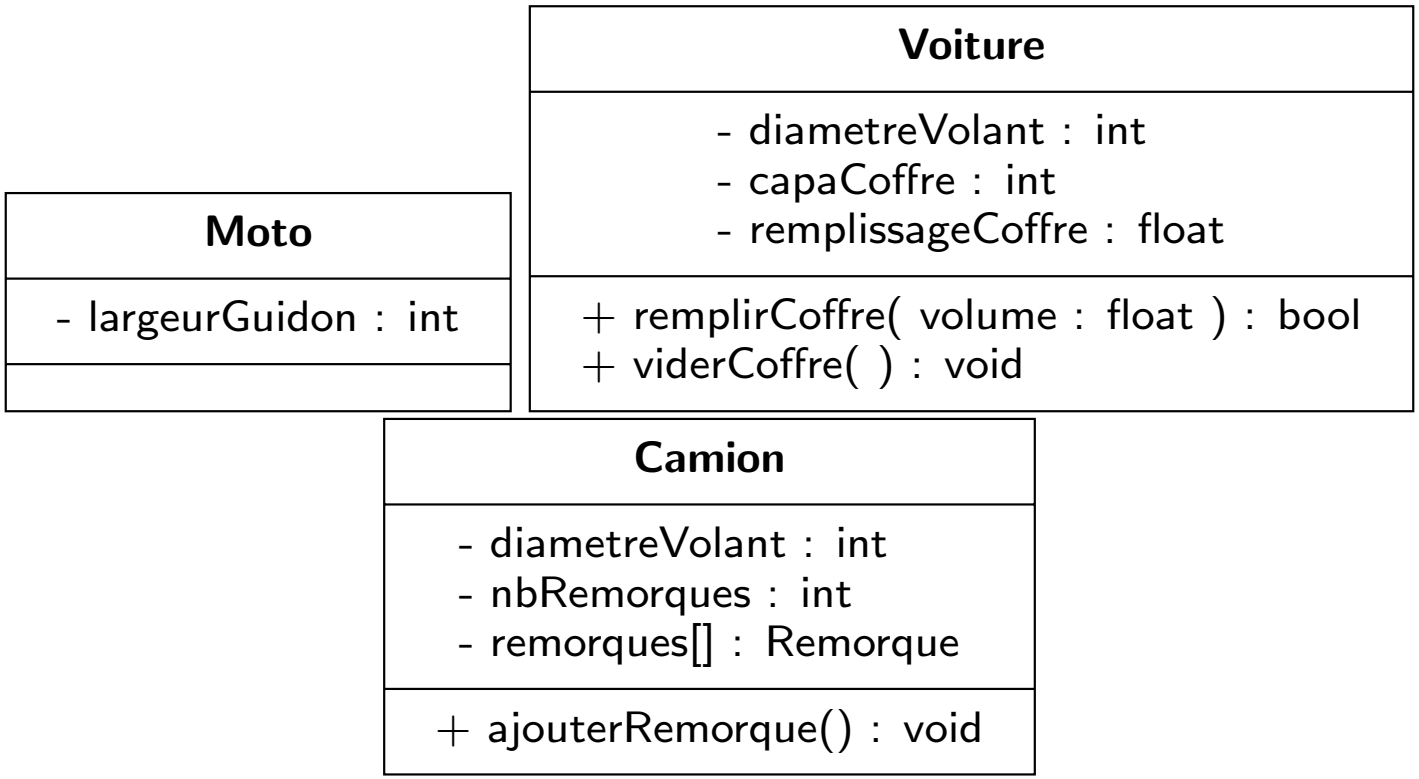
Vehicule
+ nbRoues : int - roues[] : Roue + consommation : int - niveauReservoir : float - garee : booleen - vitesse : int
+ avancer(vitesse : int, distance : int) : bool + arreter() : void + garer() : void + fairePlein() : void

Avantage : factorisation de code, abstraction d'une partie des fonctionnalités d'une classe

Héritage : classe mère et classes filles

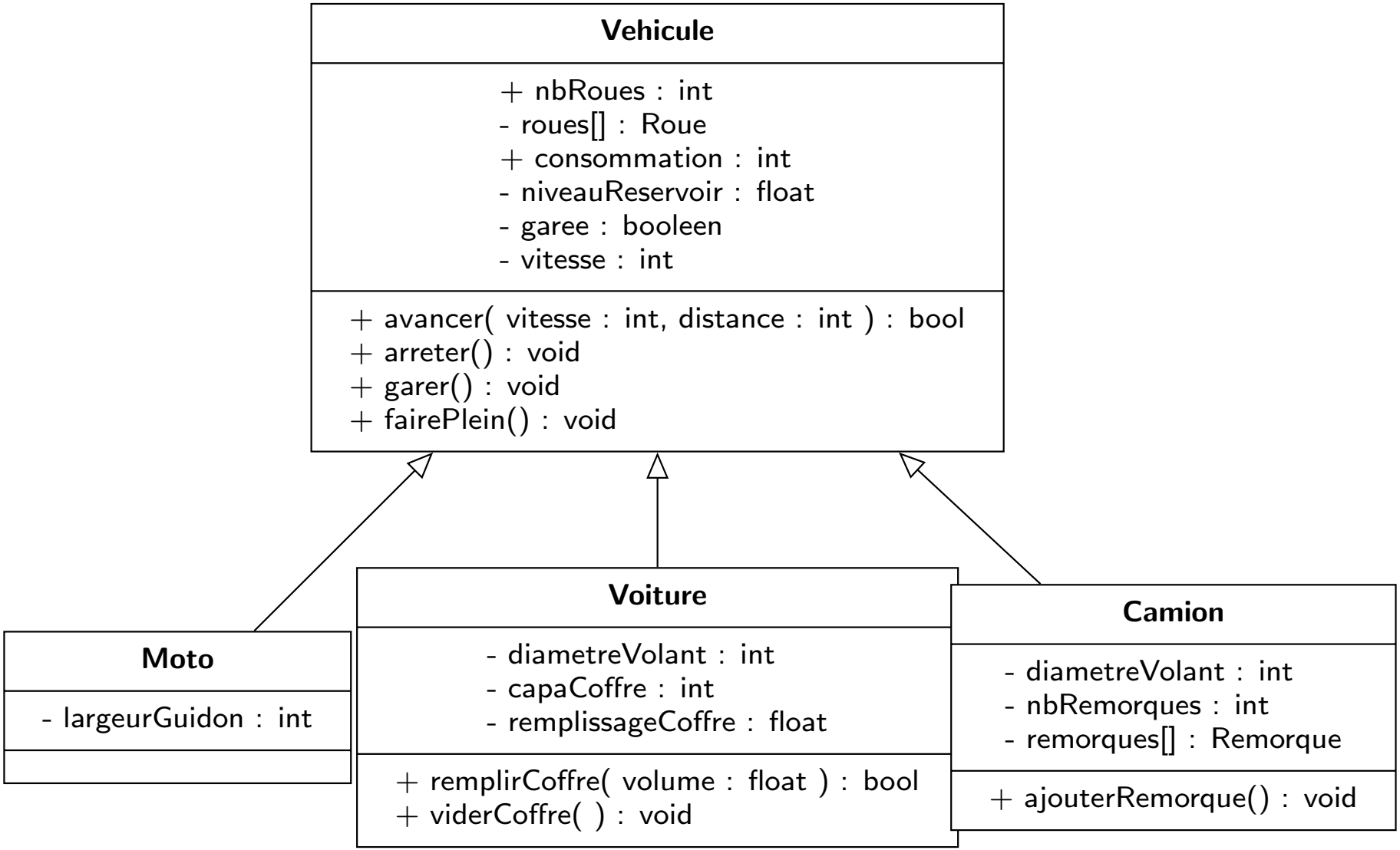
Accès aux attributs de la classe Vehicule depuis les classes Moto, Voiture et Camion par **héritage**

- La définition des classes Moto, Voiture et Camion ne contiennent que les attributs et les méthodes qui leur sont propres
- Mais on peut toujours appeler la méthode garer() d'une Moto



On ne peut hériter que d'une seule classe !

Héritage


Vehicule
+ nbRoues : int
- roues[] : Roue
+ consommation : int
- niveauReservoir : float
- garee : booleen
- vitesse : int
+ avancer(vitesse : int, distance : int) : bool
+ arreter() : void
+ garer() : void
+ fairePlein() : void
Moto
- largeurGuidon : int
Voiture
- diametreVolant : int
- capaCoffre : int
- remplissageCoffre : float
+ remplirCoffre(volume : float) : bool
+ viderCoffre( ) : void
Camion
- diametreVolant : int
- nbRemorques : int
- remorques[] : Remorque
+ ajouterRemorque() : void

Classe abstraite

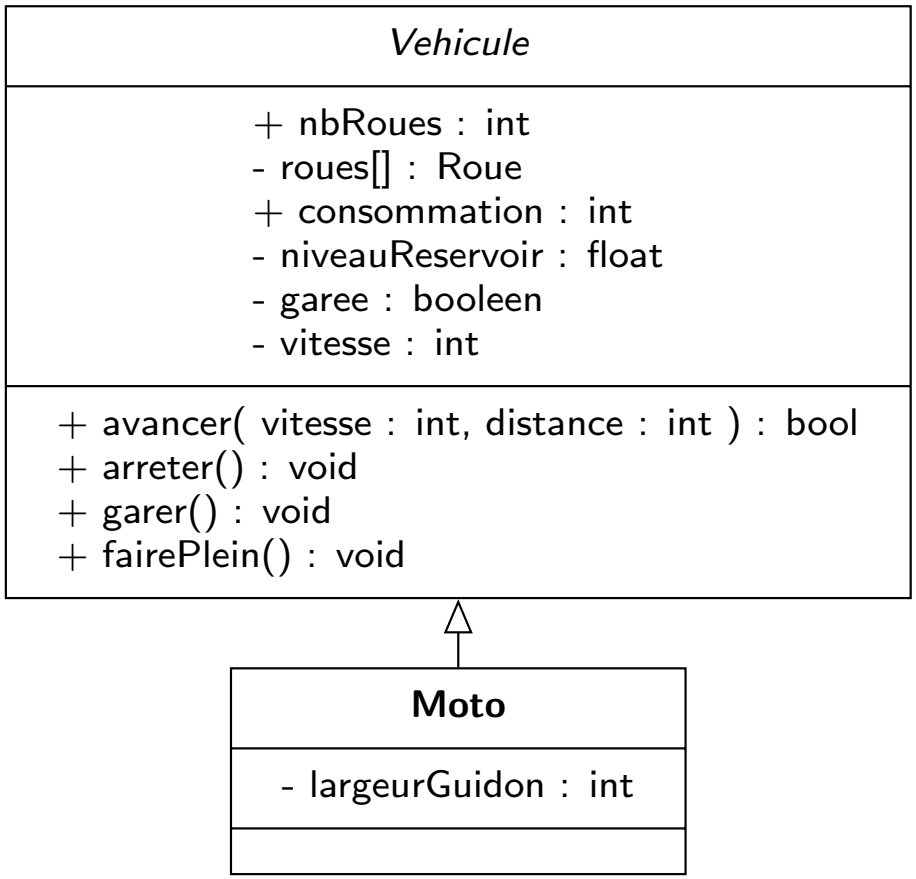
Définition

Une **classe abstraite** est uen classe dont il n'est pas possible d'instancier des objets.

Exemple : la classe Vehicule peut être une classe abstraite

- Cela n'a pas de sens d'instancier un objet de la classe Vehicule

Son nom apparait sur le diagramme de classes en *italique*.
Intérêt : ne pas instancier par erreur des objets d'une classe qui n'a pas de sens en elle-même. Utilisé uniquement dans l'arbre d'héritage.



Interfaces

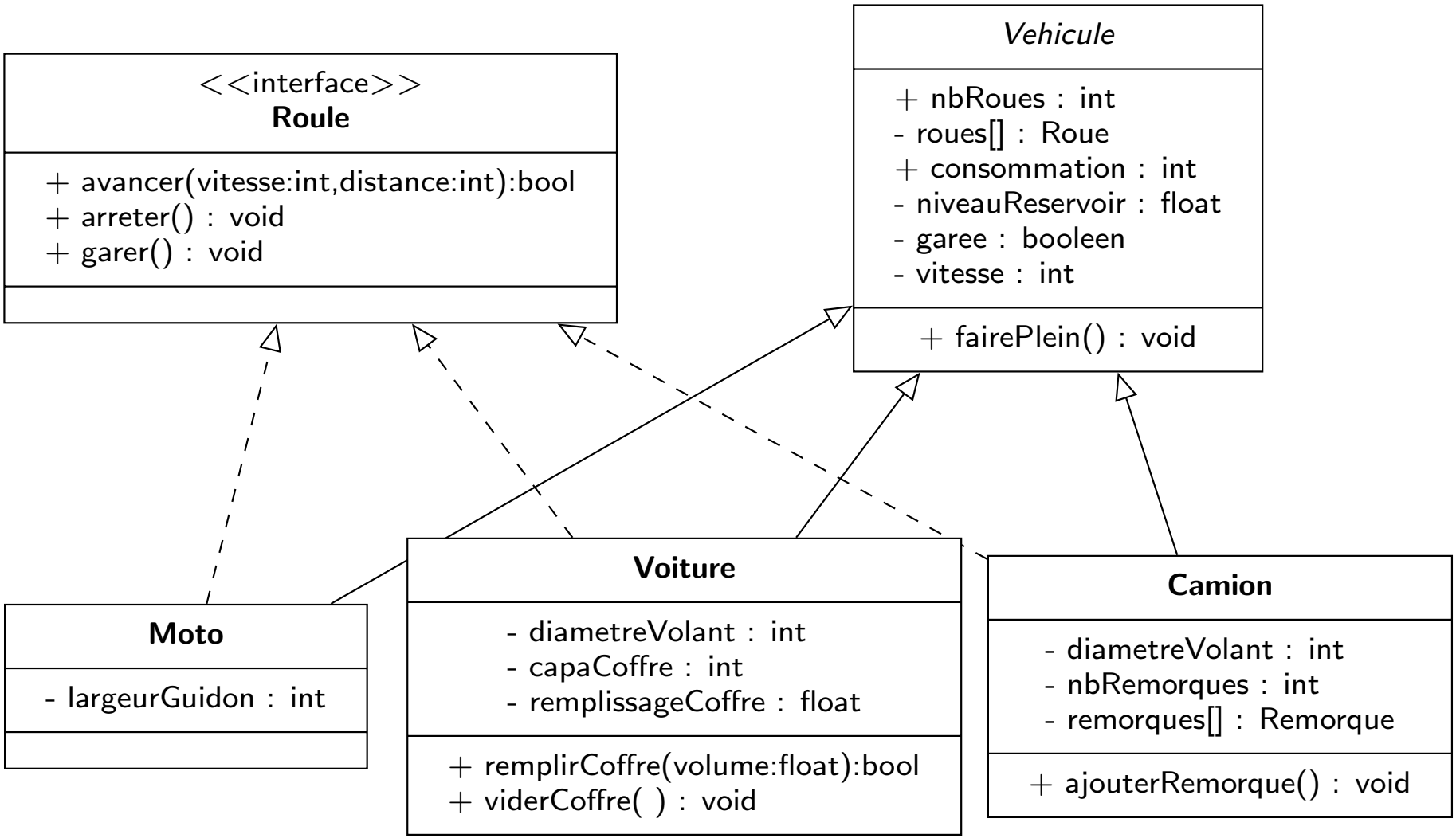
Définition

Une **interface** est une classe qui ne contient **que des méthodes**, ne donne **que leurs prototypes** et n'est pas instanciable.

Une interface définit un **contrat de comportement** : toutes les classes qui l'instancient doivent respecter de contrat et suivre le comportement défini (prototype des méthodes).
Intérêt : possibilité d'implémenter la même chose de différentes façons dans plusieurs classes différentes

- Toutes les classes implémentant une interface suivent le même comportement
- On peut instancier un objet de n'importe quelle classe, on sait quel sera son comportement

Interfaces (suite)



Propagation des données dans l'arbre d'héritage

3eme possibilité pour la portée des variables : **protégé**

- Un attribut ou une variable protégé n'est visible **que dans l'arbre d'héritage**
- Ses classes filles peuvent y accéder, pas les autres
- On le note # dans l'arbre d'héritage

On a donc en tout trois possibilités :

- les attributs et méthodes privés, représentés en précédant leur nom d'un signe –
  - Visibles uniquement dans la classe qui les définit
- les attributs et méthodes publics, représentés en précédant leur nom d'un signe +
  - Visibles dans toutes les classes
- les attributs et méthodes protégés, représentés en précédant leur nom d'un signe #
  - Visibles uniquement dans l'arbre d'héritage