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

Cours n°3
Introduction à Java

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

IUT de Villetaneuse, département R&T

2011 – 2012

Compilation

Compilation ?

- Java n'est pas réellement compilé
- Génération d'une représentation intermédiaire : le **bytecode**

Avantage : portabilité !

- Interprété par la machine virtuelle Java

Compilation :

- Compilateur : `javac`
- On lui passe le fichier qui contient la défition de la méthode `main()` (point d'entrée dans le programme)
- Il va chercher les éventuels autres fichiers tout seul

```
$ javac Toto.java
```

Exécution

Le compilateur génère des fichiers `.class`

- Un fichier par classe
- Bytecode des classes

Exécution dans la machine virtuelle Java

```
$ java Toto
```

Attention :

- Nom de la classe contenant la méthode `main()`
- **PAS DE .CLASS À LA FIN**

Point d'entrée dans le programme

Fonction `main()` : attention au prototype

```
public static void main( String[] args )
```

- Les arguments de la ligne de commande sont récupérés dans un tableau de String
- Le premier argument porte le numéro 0 (contrairement au C)

Particularités :

- Il s'agit d'une **procédure** : type **void**
  - Donc elle ne retourne rien
- Elle doit être publique
  - Déclaration avec le mot-clé **public**
- Une seule méthode `main()` pour toute la classe
  - Déclaration avec le mot-clé **static**

Première classe

```
1  class Toto {
2
3      public static void main( String[] args ){
4          System.out.println( "Hello World ! " );
5      }
6
7  }
```

On déclare une classe appelée `Toto`

- Ligne 1 : `class Toto`
- Bloc commençant par { et se terminant par }

Une seule méthode, pas d'attributs

- Ligne 3 à 5 : méthode `main()`

Affichage : `System.out.println()`

- Appel à la classe `System`
- qui contient un champ `out` (la sortie standard)
- qui contient une méthode `println()`

Variables

Types de base :

- Entiers
  - short, int, long
- Réels (nombres à virgule)
  - float, double
- Caractères
  - char (Unicode, 2 octets)
- Booléens
  - boolean

Cas des chaînes de caractères

- On utilise la classe String
- Exemple : String chaine = "Ma chaîne de caractères";

Transtypage

- Implicite (donc dangereux) : `int x = 3.14159;`
- Comme en C : `int x = (int) 3.14159;`
- Autre possibilité : `int x = int( 3.14159 );`

Tableaux

Déclaration d'un tableau de taille fixe :

```
1 int tab[5];      /* Tableau de taille 5 */
```

Si la taille est inconnue à la compilation :

```
1 int tab1[];
2 int [] tab2;    /* Équivalent */
```

Plus tard, allocation du tableau avec sa taille :

```
1 tab1 = new int[5]; /* pour 5 entiers */
```

Remplissage du tableau et accès à ses champs :

```
1 int tab[5];
2 tab = { 0, 1, 2, 3, 4 };
3 tab[0] = 0;
```

Affichages et saisies

Affichage :

- Avec retour à la ligne à la fin : `System.out.println()`
- Sans retour à la ligne : `System.out.print()`
- Argument : chaîne de caractères (String)
- Concaténation avec des variables : $+$
  - System.out.println("i vaut " $+$ i);
  - OK pour les types de base
  - Sinon : utilisation d'une méthode `toString()`

Saisie au clavier :

- Très pénible !!
- Utilisation d'une classe annexe Clavier (fournie)

Tests

Syntaxe :

```
1  if( condition ) {
2        /* bloc d'instructions à exécuter
3           si la condition est vérifiée   */
4  } else {
5        /* bloc d'instructions à exécuter
6           dans le cas contraire (facultatif) */
7  }
```

La condition est une expression qui retourne un booléen

- Une comparaison entre deux variables : `a > b`
- Une opération booléenne : `(a > b) && (a > c)`
- Toute expression retourne un booléen (fonction qui retourne un booléen, etc...)

Tests : exemple

```
1  int i;
2  i = 3;
3  if( i > 0 ) {
4    System.out.println( "i est positif et vaut " + i );
5  } else {
6    if( i == 0 ) {
7      System.out.println( "i est nul" );
8    }
9  }
```

Test avec plusieurs possibilité

Test sur une variable avec **switch**

- Plusieurs égalités testées successivement (les **case**)
- Définition d'une série d'instructions **pour chaque case**
- Fin de la série d'instructions avec **break**

```
1  int valeur;
2  valeur = 42;
3  switch( valeur ){
4    case 0:
5      System.out.println( "c'est nul" );
6      break;
7    case 1:
8      System.out.println( "valeur 1" );
9      break;
10   case -1:
11     System.out.println( "valeur -1" );
12     break;
13   default:
14     System.out.println( "autre valeur  " + valeur );
15     break;
16 }
```

Boucle for

Boucle **Pour**. Syntaxe :

```
1        for( initialisation du compteur ;
2             condition d'arrêt ;
3             modification du compteur ) {
4           bloc d'instructions
5        }
```

Le compteur doit être un entier.

Exemple :

```
1  int i;
2  for( i = 0 ; i < 3 ; i++ ) {
3      System.out.println( "valeur de i : " + i );
4  }
```

Boucle for (suite)

On peut définir n'importe quelle opération de modification du compteur.
Exemple avec un pas négatif :

```
1 int i;
2 for( i = 10 ; i > 0 ; i = i - 2 ) {
3   System.out.println( "valeur de i : " + i );
4 }
```

Affichage :

```
valeur de i : 10
valeur de i : 8
valeur de i : 6
valeur de i : 4
valeur de i : 2
```

Boucle **while**

Boucle **Tant que**. Syntaxe :

```
1 while( condition ) {
2   actions;
3 }
```

- Attention à la condition : si elle est toujours vérifiée, boucle infinie...
- Si la condition n'est jamais vérifiée on n'entre jamais dans la boucle : on teste la condition **puis on éxécute le bloc si nécessaire**

Exemple :

```
1 float i;
2 i = 0;
3 while( i < 5 ) {
4   System.out.println( "Valeur : " + i );
5   i++;
6 }
```

Boucle do ... while

Boucle **Faire... tant que**. Syntaxe :

```
1  do {
2      actions;
3  } while( condition );
```

- Attention à la condition : si elle est toujours vérifiée, boucle infinie...
- Le bloc est exécuté **au moins une fois** : on teste la condition **après exécution du bloc**

Exemple :

```
1  i = 0;
2  do {
3     System.out.println( "Valeur : " + i );
4     i++;
5  } while( i < 5 );
```

Différence entre **while** et **do ... while**

Boucle **while** :

- On évalue la condition **avant** d'exécuter le bloc
- Si la condition est réalisée, on exécute le bloc
- Si la condition n'est jamais réalisée, on n'exécute pas le bloc du tout

Boucle **do ... while** :

- On évalue la condition **après** avoir exécuté le bloc
- Si la condition est réalisée, on ré-exécute le bloc
- Si la condition n'est jamais réalisée, on exécute une fois le bloc et on s'arrête là

Différence entre **while** et **do ... while** (suite)

Exemple :

```
1  int i;
2  i = 0;
3  while( i == 1 ) {
4     System.out.println( "boucle while" );
5  }
6  do {
7     System.out.println( "boucle do ... while" );
8  } while( i == 1 );
```

- La première condition n'est jamais réalisée. Donc on n'exécute pas la ligne 4
- On exécute la ligne 7 puis la condition ligne 8 n'est pas réalisée donc on s'arrête là

Affichage obtenu :

boucle **do** ... **while**

Fonctions

Déclaration d'une fonction :

- Type retourné par la fonction (type de base ou classe déclarée)
- Nom de la fonction
- Arguments et leurs types

Valeur retournée renvoyée avec le mot-clé `return` + la valeur retournée

Exemple :

```
1  int calculCarreHypothenuse( int cote1, int cote2 ){
2    int resultat;
3    /* implémentation de la fonction */
4    return resultat;
5  }
```

Procédures

Déclaration d'une procédure :

- Pas de type retourné : `void`
- Nom de la procédure
- Arguments et leurs types

Exemple :

```
1  void afficherResultat( int resultat ){
2    /* implémentation de la procédure */
3  }
```

On sort de la procédure à la fin du bloc d'instruction où quand on rencontre le mot-clé `return`

Portée

Mot-clés `public`, `private` et `protected`

Exemple :

```
1  class Cercle {
2    /* Attributs */
3    private int diametre;
4    /* Méthodes */
5    private void afficherResultat( int diam, float surf ){
6      System.out.print( "Le diamètre est de " + diam );
7      System.out.println( " et la surface est de " + surf );
8    }
9    private float calculAire( int diametre ) {
10     float aire;
11     aire = float( MATH.Pi ) * ( diametre / 2 )
12                             * ( diametre / 2 );
13     return aire;
14   }
15   public void afficherAire( ) {
16     float aire;
17     aire = calculAire( diametre );
18     afficherResultat( diametre, aire );
19   }
20 }
```

Classes et objets

La classe qui contient la procédure `main()` est particulière

- Elle sert de point d'entrée dans le programme

Les autres classes sont utilisées pour créer des objets

- On les **instancie** avec le mot-clé **new**

Exemple :

```
1 Livre l = new Livre();
```

Destruction :

- Pas de destruction/libération explicite
- Le ramasse-miettes (garbage collector) détruit les objets vers lesquels il n'y a plus de références

Copie et comparaison d'objets

Attention : le langage Java fonctionne avec des **références**
Exemple :

```
1 Livre l = new Livre();
```

`l` est en fait une référence vers un objet de type Livre.
Copie :

```
1 Livre l1, l2;
2 l1 = new Livre();
3 l2 = l1;
```

Ici c'est la *référence* vers l'objet pointé par `l1` qui est copiée dans `l2`. On n'a pas créé de nouvel objet.
De même, si on teste l'égalité, on teste l'égalité entre les *références* et non les objets :

```
1 Livre l1, l2;
2 /* ... */
3 if( l2 == l1 ) { /* ... */}
```

Il faut donc écrire des méthodes spécifiques : `copie()` et `estEgal()`.

Constructeur et destructeur

Instanciation d'une classe avec `new` = appel à son **constructeur**.

Constructeur

Le **constructeur** d'une classe est une méthode (procédure) portant **le même nom que la classe**, appelée lors de la création d'une instance de cette classe et pouvant recevoir des paramètres (exemple : initialisation des attributs)

Il doit évidemment être public

```
1  class Cercle {
2    /* attributs */
3    int diametre;
4    /* Constructeur */
5    public Cercle( int _diametre ) {
6      diametre = _diametre;
7    }
8    /* Autres methodes */
9  }
```

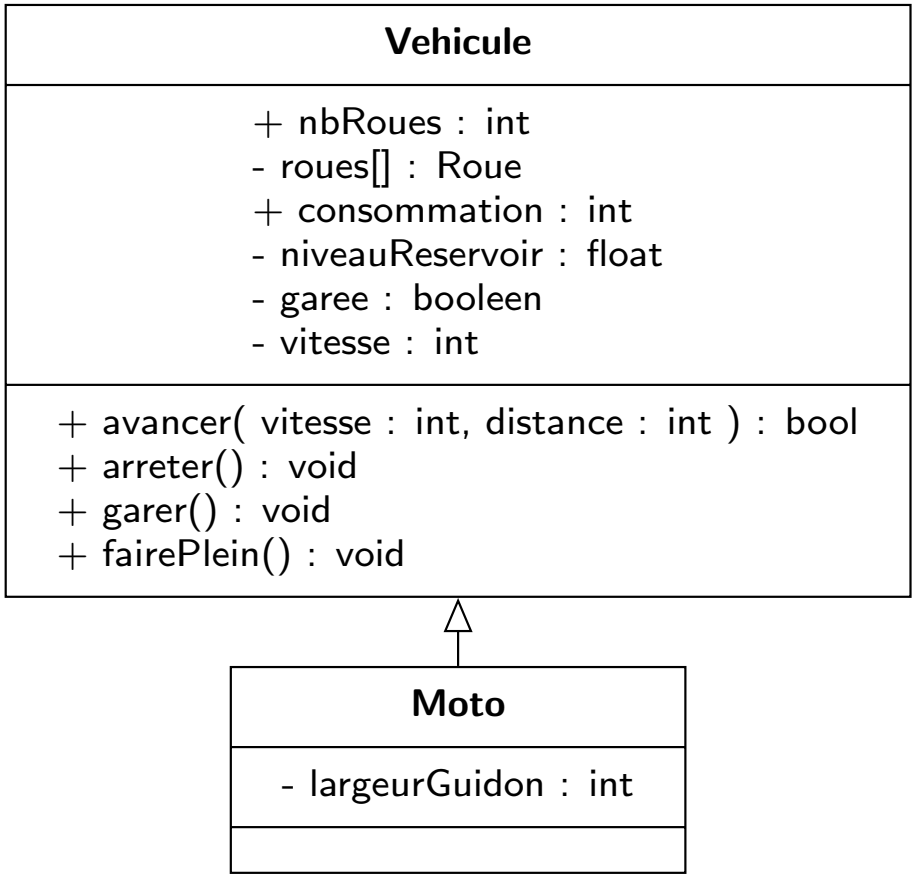
Pour forcer le passage du ramasse-miettes à un instant donné : appel à la méthode `finalize()`

- Possibilité de la définir pour une classe
- Sinon : utilisation de la méthode de la classe `Object`

Héritage

Déclaration de la classe fille avec le mot clé **Extends**

```
class Vehicule {
  /* ... */
}
class Moto Extends Vehicule {
  /* ... */
}
```



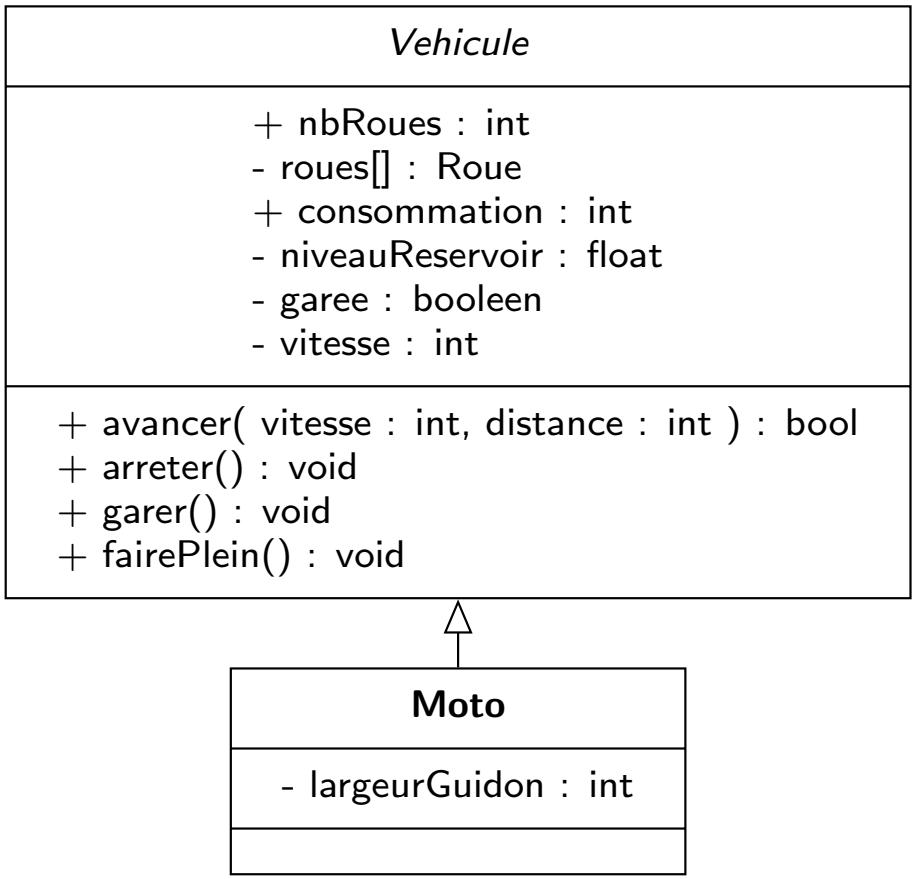
Accès aux attributs et méthodes depuis la classe fille :

- Ses propres champs : mot-clé **this**
  - Exemple : `this.largeurGuidon`
- Champs de la classe mère : mot-clé **super**
  - Exemple : `super.consommation`
  - Constructeur : `super()`

Classe abstraite

Déclaration de la classe abstraite avec le mot-clé **abstract**.

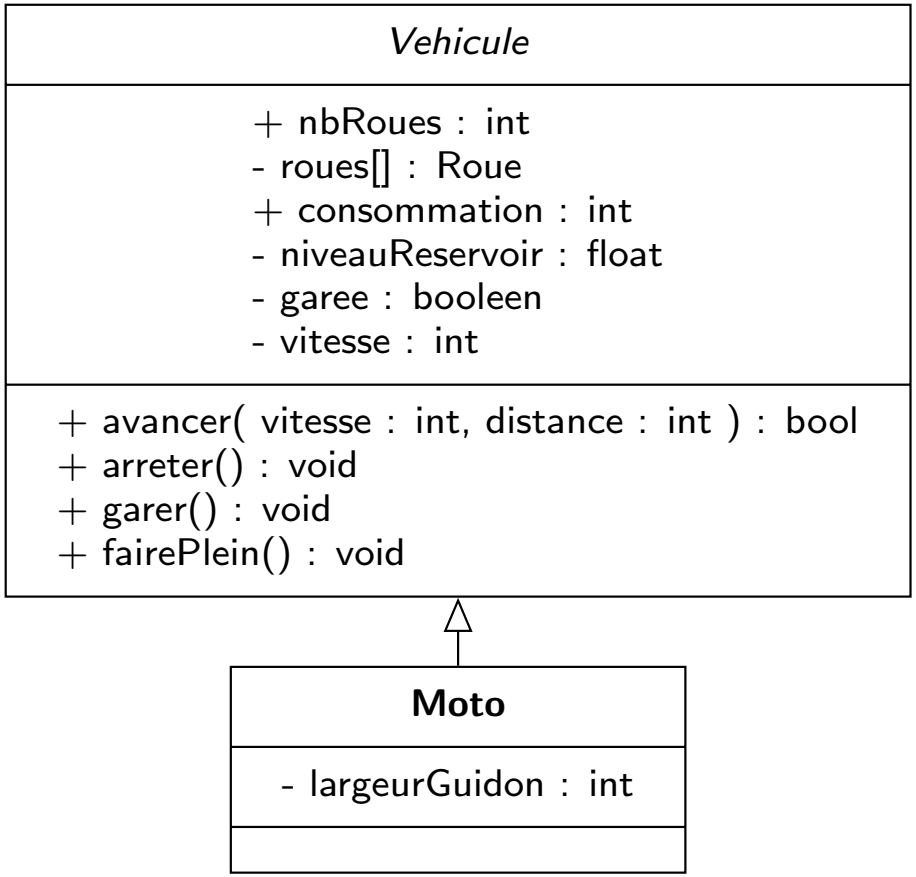
```
abstract class Vehicule {
  /* ... */
}
class Moto Extends Vehicule {
  /* ... */
}
```



Méthodes abstraites

Une **méthode abstraite** est une méthode d'une classe abstraite qu'on déclare sans en donner l'implémentation. Les classes filles donnent son implémentation qui leur est propre à chacune.
Intérêt : l'implémentation d'une méthode d'une classe générale (la classe mère) peut différer selon la spécialisation qui en est faite (les classes filles).

```
abstract class Vehicule {
   /* ... */
   public void garer() {
      /* ... */
   }
   public abstract void arrêter();
}
class Moto Extends Vehicule {
   /* ... */
   public void arrêter() {
      /* ... */
   }
}
```



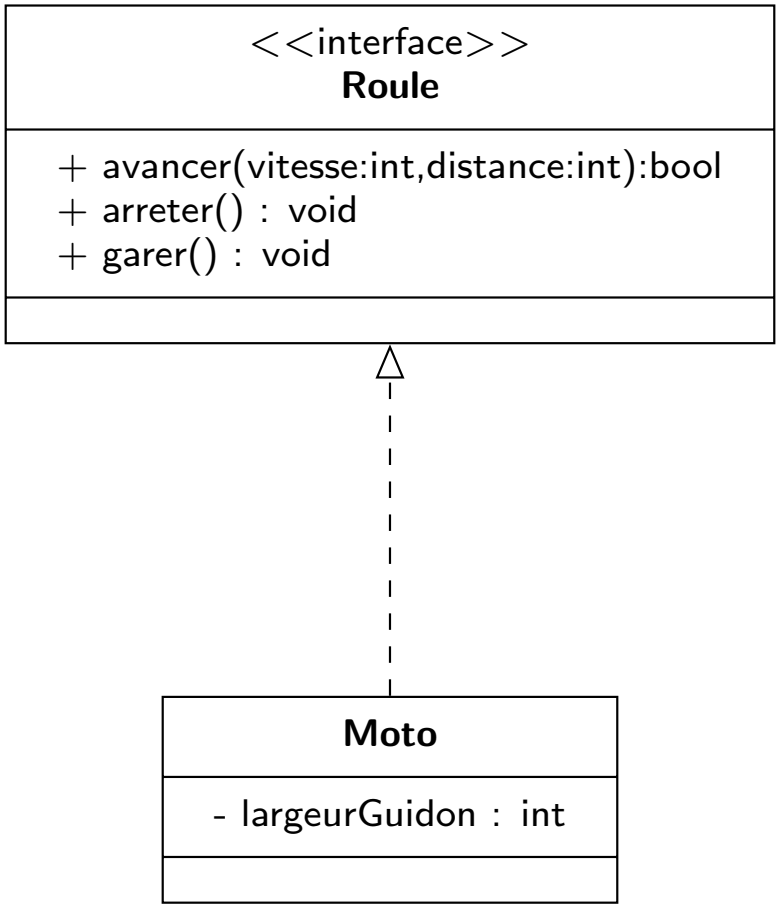
Définition des méthodes abstraites dans la classe abstraite avec le mot-clé **abstract**

- Implémentation dans les classes filles

Interface

- Déclaration d'une interface avec le mot-clé **Interface**
- Déclaration d'une classe qui l'implémente avec le mot-clé **implements**

```
public Interface Roule {
   /* ... */
}
class Moto implements Roule {
   /* ... */
}
```



Les membres d'une interface sont **toujours publics** : on n'a pas forcément besoin de spécifier leur visibilité.