**Cours Programmation Orientée Objet**

**Chapitre 1 : Les bases de la POO**

1. **Introduction :** (C’est quoi la POO ?)

L’orientée Objet et un paradigme de programmation. Le mot paradigme signifie une représentation du monde dans le sens large. C’une manière de voir les choses, un modèle avec lequel on interprète les observations.

Autrement dit, un paradigme de programmation est une *façon de concevoir* le code informatique, un moyen de *formuler* le problème dans un code informatique, d’imaginer sa solution et de l’implémenter.

Il existe plusieurs nombres de paradigmes de programmation. Parmi les paradigmes courants on peut citer les paradigmes suivants (cette liste n’est pas exhaustive) :

* Le paradigme Procédural
* Le paradigme Orienté Objet
* Le paradigme Fonctionnel
  1. **Programmation Orientée Objet :**

En programmation OO le programme est conçu avec des briques logiciels appelées ***Objets***. L’*interaction* entre ces objets via leurs *relations* permet de réaliser les fonctionnalités attendues.

1. **Concepts fondamentaux de la POO**

Cette section abordera sur les concepts fondamentaux de la programmation orientée objet, tel que les classes et les objets, la modularité et la réutilisabilité.

* 1. **Programmation procédurale vs programmation par objet**

Dans la programmation procédurale les données sont passées entre les procédures ce qui présenté schématiquement sur la Figure 1. Une procédure implémente des algorithmes de calcul et peut générer des paramètres de sortie qui sont ensuite utilisés comme des paramètres d’entrés dans une autre procédure.

Figure 1 Programmation procédurale

Par contre, dans la programmation orientée objet, les données et les algorithmes sont intégrés dans les objets (voir Figure 2). Chaque objet contient un peu de donnée et un peu d’algorithme. On utilise le terme *attribut* pour désigner les données de l’objet et le terme *méthode* pour désigner les algorithmes. Le programme est construit comme une interaction entre ces différents objets.

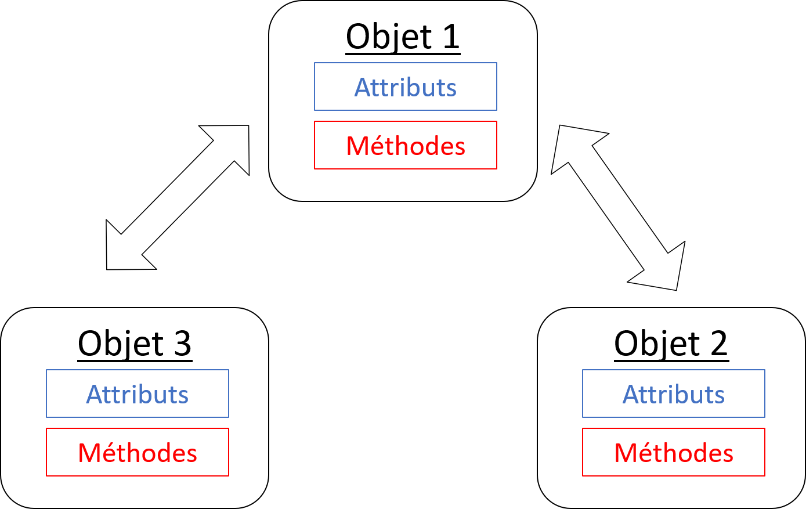


Figure 2 Programmation Orientée Objet

Pour une perception humaine, il est plus facile de décrire la réalité en termes d’objets, plutôt qu’en termes de procédures et de données. Le tableau ci-dessous résume une comparaison entre la programmation OO et la programmation procédurale.

|  |  |  |
| --- | --- | --- |
|  | Programmation Procédurale | Programmation Orientée Objet |
| Programmes | Le programme principal est divisé en petites parties selon les fonctions. | Le programme principal est divisé en petit objet en fonction du problème. |
| Les données | Chaque fonction contient des données différentes. | Les données et les fonctions de chaque objet individuel agissent comme une seule unité. |
| Accès | Aucun spécificateur d’accès n’est utilisé. | Les spécificateurs d’accès public, private, et protected sont utilisés. |
| La communication | Les fonctions communiquent avec d’autres fonctions en gardant les règles habituelles. | Un objet communique entre eux via des messages. |
| Contrôle des données | La plupart des fonctions utilisent des données globales. | Chaque objet contrôle ses propres données. |
| Importance | Les fonctions ou les algorithmes ont plus d’importance que les données dans le programme. | Les données prennent plus d’importance que les fonctions du programme. |
| Masquage des données | Il n’y a pas de moyen idéal pour masquer les données. | Le masquage des données est possible, ce qui empêche l’accès illégal de la fonction depuis l’extérieur. |
| Exemples | Pascal, Fortran | PHP5, C ++, Java. |

Tableau 1 Programmation Orientée Objet vs Programmation Procédurale

* 1. **Réutilisation du code**

La réutilisation du code est l’action de reprendre un code logiciel existant pour le réemployer, éventuellement en l'adaptant, dans un contexte présentant certaines similarités avec le programme d'origine.

La réutilisation procure des avantages tels un développement plus rapide, une augmentation de la qualité du code, des changements plus faciles à opérer, une meilleure productivité, une réduction des coûts de maintenance, des délais de livraison plus courts, une complexité réduite et une cohérence accrue des applications et architectures logicielles, la réduction des défauts et des risques et, en bout de ligne, une plus grande agilité des systèmes.

* 1. **Introduction à la modularité**

La modularité est le processus de division d’un tout en plusieurs parties bien définies, que l’on peut construire et analyser séparément et qui interagissent de manière bien précise. Elle permet de rendre les gros projets *réalisable, compréhensible, réutilisable* et *plus sûre.*

1. **Les objets et les classes**

Dans cette section nous allons présenter deux concepts fondamentaux de la programmation orientée objet, le concept classe et le concept objet.

* 1. **Notion de classe**

Une classe est une représentation abstraite d’une entité du monde réel. Elle décrit les états/propriétés et les comportements communs d’un même *type.* Les valeurs de ces états seront contenues dans les objets issues de la classe. Par exemple, Une classe voiture définit des propriétés (couleur, marque, énergie, vitesse) et des comportement (rouler, accélérer, freiner, arrêter). La classe peut être vue comme un ‘moule’ à partir duquel l’objet va être créé.

* 1. **Notion d’objet**

L’objet représente un *élément* du monde réel. Il est construit à partir d’une classe en valorisant certaines propriétés.

L’objet représente l’élément de base de la POO. L’objet est l’union de :

* Une liste de variables d’états : représente les propriétés ou les caractéristiques de l’objet. Ces variables changent de valeur durant la vie de l’objet.
* Une liste de comportements : représente ce que l’objet peut faire (traitement) ;

**Exemple.**  *Voiture* (couleur = noire, marque = Audi, énergie = diesel, vitesse = 0).

* 1. **Attributs**

L’attribut représente une propriété d’un objet. Il permet de stocker l’état courant de l’objet. Un objet peut contenir plusieurs attributs. Ils peuvent être de type primitif (entier, caractère … etc.) mais également de type plus complexe.

**Exemple.** L’attribut *vitesse* de l’objet Voiture (noire, Audi, diesel).

* 1. **Notion de méthodes (message)**

Un objet est capable à répondre à des messages. L’exécution du message (méthode) permet à l’objet de passé d’un état à un autre. En effet, On dit qu’un objet possède un comportement. Par exemple, l’objet *Voiture* (couleur = noire, marque = Audi, énergie = diesel, vitesse = 0), reçoie un message pour changer la valeur de la *vitesse*.

**Avantages :**

La programmation Orientée Objet apporte aux développeurs les moyens de relever les nouveaux défis qui sont les leurs avec :

1. Une organisation modulaire très proche à la réalité.
2. Une création, une mise au point et une maintenance des composants plus facile et plus rapide
3. La réutilisation et l’évolution des composants existants ou de composants issus de compagnie tierces.
4. Une intégration facile pour le fonctionnement dans des environnements graphiques.
5. Une logique de codage compatible avec les applications distribuées voyant leurs contenus dispatchés sur plusieurs machines.
6. Un découpage de l’application permettant un travail en équipe plus efficace et plus productif.
7. **Introduction à Java**

Java est un langage de programmation orienté objet créé en 1995 par Sun Microsystems. En 2009 a été racheté par Oracle.

* On manipule des Objets.
* Une de ses plus grandes forces est son excellente portabilité : une fois le programme créé, il fonctionnera automatiquement sous Windows, Mac, Linux, etc.
* On peut faire de nombreux types de programmes avec Java :
  + Des applications, sous forme de fenêtre ou de console ;
  + Des applets, qui sont des programmes Java incorporés à des pages Web ;
  + Des applications pour appareils mobiles, comme les smart phones, avec J2ME (Java 2 Micro Edition);

Cycle de Vie d’un programme Java :

1. Phase de codage : c’est la phase d’écriture du code source.
2. Phase de compilation :
3. Phase d’exécution :

**Sources Files**

**\*.java**

**Class Files**

**\*.class**

**Types et structure de contrôles**

* + 1. **Les types de données**

Rappelons que les programmes manipulent des variables. Une variable est un espace « conteneur » mémoire permettre de stocker une information. La nature « Type » de la variable indique l’ensemble des valeurs qu’elle pourra contenir. La variable doit être déclaré avec un nom et un type qu’elle les gardera durant toute l’exécution du programme. En java, il existe deux types de variables :

* Les types primitifs.
* Les types références « les objets ».
  + - 1. **Types primitifs**

Comme la plupart des langages de programmation, Java propose des **types** fondamentaux pour la manipulation des variables. Ce sont des containers de taille spécifique qui stockent des valeurs « primitives ». Les types primitifs comprennent les huit types basiques listés dans le tableau ci-dessous :

|  |  |  |
| --- | --- | --- |
| Type | Taille en bits | Intervalle de valeurs |
| boolean | Dépend du système | true or false |
| char | 16 bits | 0 à 65535 |
| byte | 8 bits | -128 à 127 |
| short | 16 bits | -32768 à 32767 |
| int | 32 bits | -2^31 à 2^31 |
| long | 64 bits |  |
| float | 32 bits |  |
| double | 64 bits |  |

La déclaration d’une variable s’écrit de la manière suivante :

*Type <nomVariable1> ;*

Ou

*Type <nomVariable1, nomVariable2 >;*

Exemple :

*int a  = 1 ;*

*float b = 2.0f ;*

Le langage Java est un langage fortement typé. Il interdit d’affecter une valeur d’une variable à une autre variable de type différent si cette variable n’est pas explicitement transformée.

Exemple :

*a = b ; // est interdite*

Il faut écrire *a = (int) b ;*

**Les opérateurs arithmétiques :**

« + » : permet d'additionner deux variables numériques (mais aussi de concaténer des chaînes de caractères).   
« - » : permet de soustraire deux variables numériques.   
« \* » : permet de multiplier deux variables numériques.   
« / » : permet de diviser deux variables numériques.   
« % » : permet de renvoyer le reste de la division entière de deux variables de type numérique ; cet opérateur s'appelle le modulo.

Exemples :

|  |  |
| --- | --- |
| int nbre1, nbre2, nbre3; | //déclaration des variables |
| nbre1 = 1 + 3;  nbre2 = 2 \* 6; | //nbre1 vaut 4 //nbre2 vaut 12 |
| nbre3 = nbre2 / nbre1; | //nbre3 vaut 3 |
| nbre1 = 5 % 2;  nbre2 = 99 % 8;  nbre3 = 6 % 3; | //nbre1 vaut 1, car 5 = 2 \* 2 + 1 //nbre2 vaut 3, car 99 = 8 \* 12 + 3 //là, nbre3 vaut 0, car il n'y a pas de reste |

* + - 1. **Types références « objets »**

Les types référence stockent les adresses « références » des données. Ces données sont mémorisées dans une zone mémoire dénommée le *heap* (Tas). Elles sont accessibles depuis d’autre instances de classes. Leurs vies s’arrêteront quand on s’en servira plus. C’est-à-dire, tant qu’il existe une référence active sur la zone mémoire, ces données sont maintenus. Sinon, la donnée est considérée comme inutilisable et sa destruction est géré par le *garbage collector*.

Une variable de type référence caractérise une instance de classe « objet ». On verra plus de détaille sur la création des classes et, les déclarations et instanciations des objets dans la partie classes et instanciation.

* + 1. **Structures conditionnelles**

Dans une application, l’exécution se fait d’une façon séquentielle, c’est à dire ligne par ligne. Alors que ce n’est pas le cas dans tous les scénarios par exemple la résolution d’une équation de second degré. Avec les *conditions* nous pouvons gérer les différents cas de figure sans pour autant lire tout le code.

En Java, Les mots clés réservés pour les structures conditionnelles sont : *if, if .. else, switch* et l’opérateur ternaire « ? : »

Avant de pouvoir créer et évaluer des conditions, on peut utiliser les opérateurs logiques suivants :

* **« == » :** permet de tester l'égalité ;
* **« != » :** permet de tester l'inégalité
* **« < » :** strictement inférieur
* **« <= » :** inférieur ou égal
* **« > » :** strictement supérieur
* **« >= » :** supérieur ou égal
* **« && » :** l'opérateur ET. Il permet de combiner 2 conditions. Le résultat est *vrai* si les deux conditions sont vraies. Sinon Faux
* **« || » :** l'opérateur OU. Il permet de combiner 2 conditions. Le résultat est *vrai* si une condition est vraie.

**L’instruction *if* :**

Elle permet d’exécuter un bloc d’instruction si la condition est vraie. Sa syntaxe est la suivante :

**if**(condition) {

//bloc instructions

}

Il est possible de définir plusieurs conditions à remplir avec les opérateurs vus précédemment, par exemple :

**if**((condition1) && (condition2)) {

//bloc instructions

}

**if**((condition1) || (condition2)) {

//bloc instructions

}

**L’instruction *if* … else :**

Elle permet d’exécuter le bloc d’instruction *if* si la condition est vérifiée. Dans le cas contraire, le c’est le bloc *else* qui sera exécuté. La syntaxe est la suivante :

**if**(condition) {

//bloc instructions if

} **else** {

//bloc instructions else

}

Exemple :

**if** (a > b) {

// Affiche a est plus grand que b

} **else** {

// Affiche b est plus grand que a

}

**L’instruction *switch*:**

Elle permet d’exécuter un bloc d’instructions en fonction du contenu d’une variable. Si aucune valeur ne vérifier la condition le bloc default est exécuté. Elle remplace plusieurs conditions *if … else* imbriquées. Sa syntaxe est la suivante :

**switch** (variable) {

**case** valeur1:

// bloc instruction si variable = valeur1

**break**;

**case** valeur2:

// bloc instruction si variable = valeur2

**break**;

**default**:

// bloc instruction si aucune valeur ne remplie la condition

**break**;

}

* + 1. **Structures répétitives**

Les boucles sont des structures servent à répéter l’exécution d’un bloc d’instruction jusqu’à ce que la condition ne soit pas vérifiée.

**La boucle ‘*for’*:**

La syntaxe de la boucle *for* est la suivante :

***for*** *(init compteur ; condition ; modification compteur){*

*//bloc instructions*

*}*

Elle est composée de trois parties :

* *Init compteur*: initialisation de la variable qui sert de compteur ;
* *Condition*: la condition sur laquelle la boucle s’arrête ;
* *Modification compteur*: une instruction qui incrémente/décrémente le compteur.

Exemple : la boucle ci-dessous affiche le carré des nombres entiers de 0 à 9

***for*** *(****int*** *i = 0; i < 10; i++) {*

*//afficher i\*i*

*}*

**La boucle ‘*while’*:**

La syntaxe de la boucle ***while*** est la suivante :

**while** (condition) {< *bloc instructions*>}.

L’exécution de la boucle ***while*** suit les étapes suivantes :

1. Evaluation de la condition, qui renvoie une valeur booléenne. Si celle-ci est vrai allez à l’étape b sinon on passe à l’étape d ;
2. Exécution du bloc d’instructions ;
3. Retour à l’étape a ;
4. La boucle est terminée et le programme continue son déroulement.

Exemple : x = 0 ; y = 10 ; **while** (x < y) {x++ ;}

**La boucle ‘do … *while’*:**

La boucle *‘do … while’* s’exécute comme suit :

1. Exécution du bloc d’instructions ;
2. Evaluation de la condition, qui renvoie une valeur booléenne. Si celle-ci est vrai allez à l’étape a sinon on passe à l’étape c ;
3. La boucle est terminée et le programme continue son déroulement.

La syntaxe de la boucle ***while*** est la suivante : **do**{//bloc d'instructions} **while**(condition);

* + 1. **Les Tableaux**

Un tableau est un conteneur qui stocke un nombre fixe de valeurs d'un même type. La longueur d'un tableau est établie lorsque le tableau est créé. Chaque case d'un tableau est appelée un élément, et chaque élément est accessible par son index numérique.

**Création, initialisation et accès à un tableau**

La déclaration d’un tableau, en Java, peut se faire de façon suivante :

* <Type du tableau > [<taille du tableau> ] <nom du tableau> ; ou
* <Type du tableau > <nom du tableau> [ ] = {<contenu du tableau>}

Cependant, la convention décourage cette dernière forme.

Une façon de créer un tableau consiste à utiliser l’opérateur ***new***. L'instruction suivante alloue un tableau avec suffisamment de mémoire pour 10 éléments entiers et affecte le tableau à la variable *tableauEntier*.

**int**[] tableauEntier = **new** **int**[10];

Les lignes suivantes attribuent des valeurs à chaque élément du tableau :

tableauEntier [0] = 100; //initialiser le premier élément du tableau

tableauEntier [1] = 200; //initialiser le deuxième élément du tableau

tableauEntier [2] = 300; //et ainsi de suite

on peut également utiliser la syntaxe suivante pour créer et initialiser un tableau :

**int**[] tableauEntier = {

100, 200, 300,

400, 500, 600,

700, 800, 900, 1000

};

Dans cette situation, la longueur du tableau est déterminée par le nombre de valeurs fournies entre accolades et séparées par des virgules.

Nous pouvons également déclarer un tableau de tableaux (également appelé tableau multidimensionnel) en utilisant deux ou plusieurs ensembles de crochets, par exemple : **int**[][] matrice ;

* 1. **Classes en Java**

Rappelons qu’une classe est une structure informatique représentant les principales caractéristiques d'un élément du monde réel grâce :

* À des variables, qui représentent les divers attributs de l'élément que nous souhaitons  
  utiliser ;
* À des méthodes, qui permettent de définir les comportements de nos éléments.

En Java, une classe est déclarée comme suit :

**public** **class** MyFirstClass {

//déclaration des attributs

//déclaration des constructeurs

//déclaration des méthodes

}

Le corps de la classe (la zone entre les accolades) contient tout le code qui fournit le cycle de vie des objets créés à partir de la classe. En générale la définition d’une classe peut inclure ces composants, dans l'ordre :

1. La déclaration de la classe en utilisant le mot clé ***class*** suivi par le nom de la classe[[1]](#footnote-1).
2. Les attributs (variables d’instances)
3. Un ou plusieurs constructeurs
4. Les méthodes

Exemple : **public** **class** Voiture { . . . }

* 1. **Les attributs**

Les attributs représentent les variables d’instances. Ce sont eux qui définiront les caractéristiques de notre objet. Les déclarations des attributs sont composées de trois composants, dans l'ordre :

modificateur type nomAttribut

1. Zéro ou plusieurs modificateurs, tels que *public* ou *privé*.
2. Le type de l’attribut.
3. Le nom de l’attribut.

Exemple : La classe Voiture utilise les lignes de code suivantes pour définir ses champs :

**public** **class** Voiture {

public int vitesse ;

public int transmission ;

public String marque ;

}

Notons qu’il existe deux autres types variables :

* Les variables de classe : celles-ci sont communes à toutes les instances de votre classe.
* Les variables locales et les paramètres : ce sont des variables que nous utiliserons à l’intérieurs des méthodes
  1. **Méthodes**

Une méthode représente le traitement qu’on peut effectuer sur les attributs d’un objet. Voici la syntaxe de déclaration de méthode typique :

<modificateur> <type de retour> <nomMethode> (<paramètres>){

//Corps de la méthode

**return** resultat ;

}

Une méthode est définie pas sa **signature** qui spécifie sa portée, son type de retour, son nom et ses paramètres entre parenthèses. La signature est suivie d’un bloc de code que l’on appelle le **corps** de méthode. Dans le cas où la méthode ne renvoie pas des résultats on écrit le nom de la méthode précédé par le mot clé *void*.

Exemple : nous ajoutons deux méthodes à la classe voiture ; la méthode getVitesse et la méthode accélérer :

**public** **class** Voiture {

public int vitesse ;

public int transmission ;

public String marque ;

public void accelerer (int v ){

vitesse = vitesse + v;

}

public int getVitesse (){

**return** vitesse ;

}

}

**La méthode *toString***

La signature de la méthode toString est la suivante :

**public** String toString( ) {

**return** une chaine de caractère représentant l’objet

}

La méthode *toString*() sert à transformer un objet de n'importe quel type en une chaine de caractère. Elle renvoie une chaine de caractère représentant l’objet.

**La méthode *main***

La signature de la méthode main est la suivante :

**public** **static** **void** main(String[] args) {//quelque chose à exécuter}

La méthode *main* est le point de départ/entrée pour exécuter un programme Java. La méthode *main* peut contenir du code pour exécuter ou appeler d’autres méthodes, et elle peut être placée dans n’importe quelle classe faisant partie d’un programme. Il ne peut y’avoir qu’une seule méthode *main* active par projet Java.

* + 1. **Surcharge des méthodes**

La surcharge de méthode est un concept qui permet à une classe d’avoir plusieurs méthodes portant le même nom, si leurs listes d’arguments sont différentes.

**Deux façons de surcharger une méthode :**

Afin de surcharger une méthode, les listes d’arguments des méthodes doivent différer dans l’une ou l’autre :

* Nombre de paramètres : les méthodes doivent avoir un nombre différent de paramètres. Par exemple : somme (int, int) et somme (int, int, int).
* Type de données de paramètres : les types des paramètres doivent être différents. Par exemple, somme (int, int) et somme (double, double).
  1. **Constructeur par défaut et autres constructeurs**

Un constructeur est une méthode d'instance qui va se charger de créer un objet et, le cas échéant, d'initialiser ses variables de classe. Cette méthode a pour rôle de signaler à la JVM qu'il faut réserver de la mémoire pour l’objet et de retourner l’adresse de l’emplacement mémoire choisi.

La déclaration d’un constructeur doit respecter les règles suivantes :

1. Le(s) constructeur(s) d’une classe doit (doivent) avoir le même nom que la classe.
2. Le constructeur ne possède pas un type de retour ;
3. Une classe doit avoir au moins un constructeur ; dit constructeur par défaut.
4. Le constructeur par défaut ne contient aucun paramètre.
5. Une classe peut avoir plusieurs constructeurs.

Exemple : nous créons deux constructeurs pour la classe voiture, dont un sans paramètres et le 2éme avec paramètres.

**public** **class** Voiture {

public int vitesse ;

public int transmission ;

public String marque ;

public Voiture (){ //constructeur sans paramétres

vitesse = 0 ;

transmission = 0 ;

marque = "Audi";

}

public Voiture (int v, int t, String m){ //constructeur avec paramétres

vitesse = v ;

transmission = t ;

marque = m;

}

public void accelerer (int v ){

vitesse = vitesse + v;

}

public int getVitesse (){

return vitesse ;

}

}

* 1. **Objet et instanciation**

Java est un langage orienté objet : cela signifie en particulier que le programmeur peut définir un certain nombre de classes, que l’on peut voir en première approche comme des *types* ou des *structures* de données intelligentes. Une fois une classe définie, il est possible de créer des objets appartenant à cette classe. Chaque objet est appelé instance de la classe.

Pour créer un objet, il est nécessaire de :

1. Déclarer une variable dont le type est le nom de la classe à instancier.
2. Faire appel à un constructeur de cette classe en utilisant l’opérateur *new*.

Syntaxe de la déclaration : <nom de la classe> <nom de l’objet> ;

Exemple : Créer une instance de la classe voiture Voiture maVoiture ; cette instruction représente la déclaration d’une variable de type objet dont le nom est *maVoiture* qui pourra contenir un objet de type *Voiture.* L’exécution de cette ligne de code permet de créer un objet *null* (voir Figure 3).

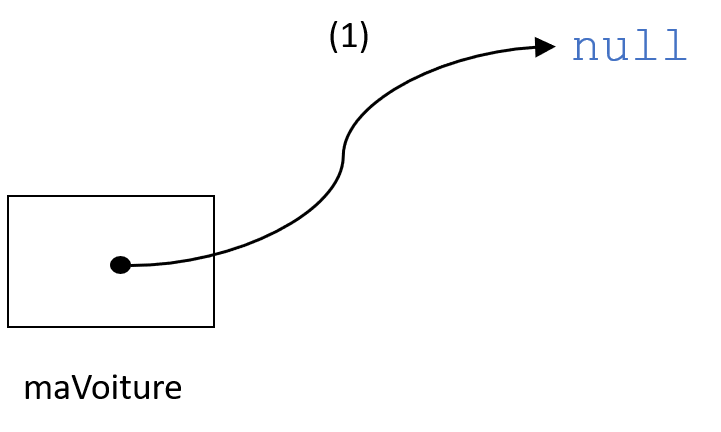


Figure 3 déclaration d'un objet

Syntaxe de l’instanciation : <nom de l’objet> = new <constructeur>

Exemple : maVoiture = new Voiture () ; l’exécution de cette ligne de code permettra de créer un nouvel objet en mémoire et retourner sa référence vers la variable maVoiture (voir Figure 4).

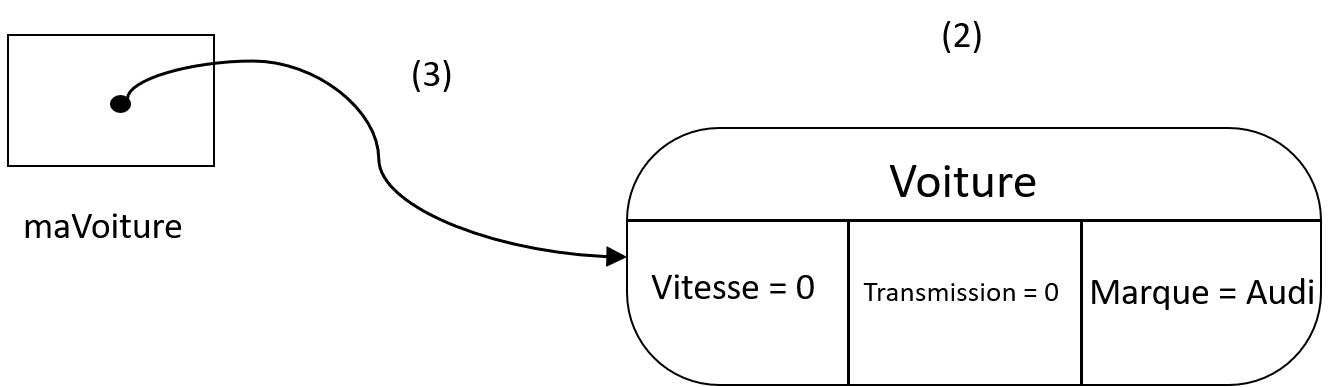


Figure 4 Création d'un nouvel objet

Pour accéder à un attribut ou une méthode associe à un objet, il faut préciser l’objet qui la contient. Le Symbole ‘.’ Sert à séparer l’identificateur de l’objet de l’identificateur de l’attribut ou la méthode.

Pour affecter une valeur à l’attribut *vitesse* de l’objet *maVoiture* on écrit :

maVoiture.vitesse = 10 ;

Pour stocker la valeur de la vitesse de l’objet maVoiture dans une variable on écrit :

int v = maVoiture.vitesse ;

Pour faire appel à la méthode *accelerer* de l’objet maVoiture on écrit :

maVoiture.accelerer(10) ;

Quelques classes utiles en java

|  |  |
| --- | --- |
| Classes | Méthodes |
| String : La classe String représente les chaînes de caractères. Toutes les chaînes littérales des programmes Java, telles que "abc", sont implémentées comme des instances de cette classe. | * charAt(int index): renvoie le caractère à l’indice index * concat(String s): concatène s à la fin de la chaine. * contains(String s): renvoie vrai si la chaine contient s * length() : renvoie la longueur de la chaine * substring(int début, int fin): renvoie une sous chaine. |
| Integer, Float, Double, Boolean…etc : Ces classes englobe une valeur du type primitif correspondant dans un objet. | * parseType(String s) : convertit une chaine de caractère vers le type adéquat(int i=Integer.parseInt("1");) |

* 1. **Destructeur**

Le destructeur est l'opposé du constructeur. Le constructeur est utilisé pour initialiser les objets tandis que le destructeur est utilisé pour supprimer ou détruire l'objet et libère la ressource occupée par l'objet.

Pour rappel, le concept de destructeur n'existe pas en Java. À la place du destructeur, Java fournit le *Garbage Collector* (ramasse miette) qui fonctionne de la même manière que le destructeur. Le garbage collector est un programme qui s'exécute sur la JVM. Il supprime automatiquement les objets inutilisés (espace mémoire qui n’a pas de référence) et libère la mémoire. Le programmeur n'a pas besoin de gérer la mémoire manuellement. Cela peut être source d'erreurs, vulnérable, et peut conduire à une fuite de mémoire.

* 1. **Package**

Un package est un espace de noms qui organise/regroupe un ensemble de classes et d'interfaces associées. D'un point de vue conceptuel, les packages peuvent être comparés aux différents dossiers d’un ordinateur. On peut conserver les pages HTML dans un dossier, les images dans un autre, et les scripts ou les applications dans un autre encore. Tandis que, les applications écrites en langage de programmation Java pouvant être composées de centaines ou de milliers de classes individuelles, il est judicieux d'organiser les choses en plaçant les classes connexes dans des packages.

L’utilité des package est multiple :

* Ils facilitent la recherche de l'emplacement physique des classes quand elles sont nécessaires (pour la compilation d'un fichier ou pour l'exécution d'un programme)
* Ils rendent très improbable qu'il y ait confusion entre des classes de même nom ;
* Ils structurent l'ensemble des classes selon une arborescence, ce qui rend beaucoup plus lisible l'ensemble ;
* Ils permettent de nuancer des niveaux de visibilité entre les classes selon qu'elles appartiennent ou non à un même paquetage.

Les noms de package prennent une forme telle que *java.monPackage.* Si la classe *MaClasse* se trouve dans le package *java.monPackage* alors son nom complet est *java.monPackage.MaClasse.* Le fichier *MaClasse.java* sera nécessairement rangé dans un répertoire de nom *monPackage* contenu par un répertoire de nom *java* (voir Figure *5*).

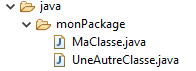


Figure 5 Exemple d'un package Java

Pour créer un *package*, on doit choisir un nom pour le package (et placer une déclaration de package portant ce nom en tête de chaque fichier source contenant la classe que on souhaite inclure dans le package.

L'instruction de *package* doit être la première ligne du fichier source. Il ne peut y avoir qu'une seule déclaration de package dans chaque fichier source.

Exemple :

**package** java.monPackage;

**public** **class** MaClasse { //...}

**Importer une classe d’un package**

Pour importer une classe spécifique dans la classe en cours, on place une déclaration *import* au début du fichier avant la définition de la classe (avant le mot clé *classe)* mais après la déclaration package, s'il y en a une. Voici un exemple comment importer la classe *MaClasse* du package créer précédemment.

**import** java.monPackage.MaClasse;

**public** **class UneAutreClasse {//...}**

Pour importer toutes les classes contenues dans un package particulier, on utilise l'instruction *import* avec le caractère générique astérisque (\*). **import** java.monPackage.\*;

La plate-forme Java fournit une énorme bibliothèque de classes (un ensemble de paquets) qu’on peut utiliser dans vos propres applications. Cette bibliothèque est connue sous le nom d'"Application Programming Interface", ou "API" en abrégé. Ses paquets représentent les tâches les plus couramment associées à la programmation générale.

|  |  |
| --- | --- |
| Package | Descriptions |
| java.lang | Un des paquets intégrés fournis par Java qui contient des classes et des interfaces fondamentales pour la conception du langage de programmation Java. Le package lang contient des classes telles que Integer, Math, etc., qui sont considérées comme les blocs de construction d'un programme Java. |
| java.util | Il contient les collections et diverses classes utilitaires. |
| [java.time](https://docs.oracle.com/javase/10/docs/api/java/time/package-summary.html) | Il fournit une API pour les dates, les heures, les instants et les durées. |
| Java.io | Il fournit des classes pour l'entrée et la sortie du système via des flux de données, la sérialisation et le système de fichiers. |
| java.net | Il Contient des classes et des interfaces qui fournissent une infrastructure puissante pour la mise en réseau en Java |
| javax.swing | Il fournit des classes tels que JButton, JTextField, JTextArea, JRadioButton pour créer des interfaces utilisateur graphiques (GUI) |

* 1. **Entrées/Sorties**

Les entrées/sorties en Java consiste en un échange de données entre le programme et une autre source, par exemple la mémoire, un fichier ou l’utilisateur lui-même. Pour réaliser cela, Java emploie ce qu’on appelle un *stream* (signifie « flux »). Ce dernier, joue le rôle de médiateur entre la source de la donnée et sa destination. Java décompose les objets traitant des flux en deux catégories :

* Les objets travaillant avec des flux d’entrée (‘in’), pour la lecture de flux ; par exemple *System.in* représente l’entrée standard.
* Les objets travaillant avec des flux de sortie (‘out’), pour l’écriture de flux ; par exemple *System.out* représente la sortie standard.

**Ecriture (sortie standard)**

Lorsque on écrit System.out.println() ; c’est-à-dire qu’on applique la méthode *println()* sur la sortie standard. La méthode *println()* permet d’écrire/afficher sur l’écran.

Exemple : System.out.println("La vitesse de la voiture est "+maVoiture.vitesse) ;

Résultats : la vitesse de la voiture est 10

**Lecture (entrée standard)**

Ici, nous allons utiliser l’entrée standard *System.in.* Donc, avant d’indiquer à Java qu’il faut lire ce qu’un utilisateur va taper sur le clavier, nous devrons instancier un objet de la classe java.util.*Scanner*. Par exemple, *Scanner clavier = new Scanner (System.in) ;*

Lecture d’un entier : pour lire une variable de type *int*  on utilise la méthode *nextInt*() de l’objet clavier de type Scanner. Par exemple, *int v = clavier.nextInt( )* ;

Lecture d’un réel : pour lire une variable de type *double*  on utilise la méthode *nextDouble*() de l’objet clavier de type Scanner. Par exemple, *double d = clavier.nextDouble ( )* ;

Lecture d’une chaine de caractère : pour lire une variable de type *String*  on utilise la méthode *nextLine*() de l’objet clavier de type Scanner. Par exemple, *String s = clavier.nextLine ( )* ;

* 1. **Exercices**

Exercice 1 : écrire un programme java qui permet de trouver le maximum entre trois variables entières.

Exercice 2 : écrire un programme java qui permet de rechercher la valeur max dans un tableau d’entier.

Exercice 3 : écrire un programme java qui permet de modéliser un étudiant.

1. Par convention, le nom d’une classe commence par une lettre majuscule. [↑](#footnote-ref-1)