Chapitre 1

Bases de la Programmation Orientée Objet

1. Introduction

La programmation procédurale a fait progresser la qualité de la production des logiciels en améliorant l'exactitude et la robustesse. Cependant, l'apparition de nouvelles exigences (notamment dans le secteur industriel) nécessitant le développement des applications complexes a mis les spécialistes dans l'obligation de revoir leurs façons de spécifier, concevoir et implémenter ces applications. Il s'agit de produire des logiciels avec des exigences de qualité qu'on tente de mesurer suivant certains critères, notamment :

- **Exactitude :** aptitude d'un logiciel à fournir les résultats voulus, dans des conditions normales d'utilisation (par exemple, données correspondant aux spécifications) ;
- **Robustesse :** aptitude à bien réagir lorsque l'on s'écarte des conditions normales d'utilisation (le logiciel est capable de bien fonctionner dans des conditions anormales et ne déclencher pas une catastrophe) ;
- **Extensibilité :** facilité avec laquelle un programme pourra être adapté pour satisfaire une évolution des spécifications ;
- **Réutilisabilité :** possibilité d'utiliser certaines parties (modules) du logiciel pour résoudre un autre problème ;
- **Portabilité :** c'est la facilité d'exécuter le logiciel sur différentes plateformes ;
- **Efficience :** cela se traduit par la bonne utilisation des ressources (temps d'exécution, taille mémoire). Autrement-dit, c'est la capacité de parvenir à un maximum de résultats avec un minimum de ressources.

Afin de répondre à ces différents besoins applicatifs, un paradigme de programmation est apparu, à savoir, la programmation orientée objet.

La programmation orientée objet tend à se rapprocher de notre manière naturelle d'appréhender le monde. En utilisant ce paradigme (POO) on doit toujours se poser la question "Sur quoi porte le programme ?". Un logiciel comporte toujours des traitements et des données. Conceptuellement parlant, la programmation structurée s'intéresse aux traitements puis aux données, alors que la conception objet s'intéresse d'abord aux données, auxquelles elle associe ensuite les traitements. L'expérience a montré que les données sont ce qu'il y a de plus stable dans la vie d'un programme, il est donc intéressant d'architecturer le programme autour de ces données.

La programmation orientée objet permet de concevoir une application sous la forme d'un ensemble de briques logicielles appelées **objets**. Chaque objet joue un rôle précis et peut communiquer avec les autres objets. Les interactions entre les différents objets vont permettre à l'application de réaliser les fonctionnalités attendues. Jusqu'à date, la POO constitue le standard actuel en matière de développement de logiciels.

2. Concepts fondamentaux de la POO

a) Petit historique de la POO

La programmation orientée objet (POO), est un paradigme de programmation informatique élaboré par les Norvégiens Ole-Johan Dahl et Kristen Nygaard au début des années 1960 et poursuivi par les travaux de l'Américain Alan Kay dans les années 1970. Les deux chercheurs Norvégien ont développé les bases de la programmation orientée objet en créant le langage Simula en 1962. Les notions de base de la POO comme les classes, l'héritage, les méthodes virtuelles, etc. furent alors créées dans ce langage pour parvenir à modéliser de façon fidèle des processus industriels complexes.

En 1980, Smalltalk est le premier langage objet proposé avec un environnement de développement graphique intégré. Smalltalk a été conçu par l'équipe de l'Américain Alan Kay au centre de recherche informatique californien de Xerox.

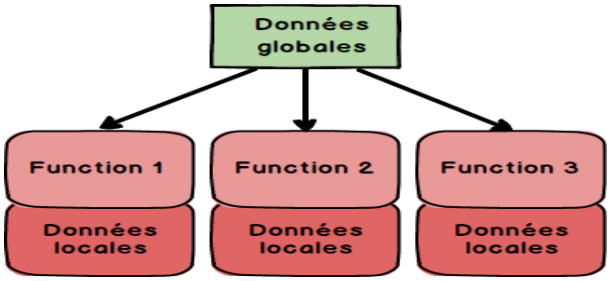
Ensuite, les principes de la POO sont appliqués dans de nombreux langages comme C++ (une extension objet du langage C créé par le Danois Bjarne Stroustrup en 1983) ? Objective C (une autre extension objet du C créé par Brad Cox en 1983 et utilisé, entre autres, par l'iOS d'Apple) ou encore Eiffel (créé par le Français Bertrand Meyer en 1986),

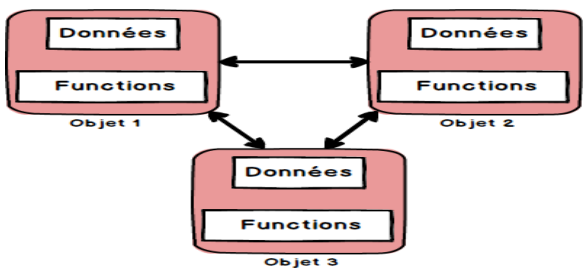
Les années 1990 ont vu l'avènement des langages orientés objet dans de nombreux secteurs du développement logiciel, et la création du langage Java par James Gosling et Patrick Naughton, employés de la société Sun Microsystems en 1995). Le succès de ce langage, plus simple à utiliser que ses prédécesseurs, a conduit Microsoft à créer au début des années 2000 la plate-forme .NET et le langage C#.

De nos jours, de très nombreux langages permettent d'utiliser les principes de la POO dans des domaines variés : Java et C# bien sûr, mais aussi PHP (à partir de la version 5), VB.NET, PowerShell, Python, etc. Une connaissance minimale des principes de la POO est donc indispensable à tout informaticien, qu'il soit développeur ou non.

b) Programmation procédurale vs programmation par Objet

Les deux figures suivantes montent la relation entre les données et les traitements dans les deux styles de programmation.





Le tableau ci-après présente une comparaison entre les deux styles de programmation, procédurale et orienté objet.

	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.
Permission	Pour ajouter de nouvelles données au programme, l'utilisateur doit s'assurer que la fonction le permet.	Le passage de message garantit l'autorisation d'accéder au membre d'un objet à partir d'un autre objet.
Exemples	Pascal, Fortran	PHP5, C ++, Java.
Accès	Aucun spécificateur d'accès n'est utilisé.	Les spécificateurs d'accès public, private, et protected sont utilisés.
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.

c) Réutilisation du code

La réutilisation du code est l'utilisation de logiciels existants, déjà testés, pour créer de nouveaux logiciels. C'est l'une des techniques inévitables du développement logiciel moderne. La réutilisation du code présente de nombreux avantages, nous citons entre autres :

- Le programmeur n'a pas besoin d'écrire autant de code.
- Meilleure structuration de code.
- Meilleure lisibilité du code.
- Réduction de l'effort de test.

d) Introduction à la modularité

La programmation procédurale a apporté une solution au problème de la répétition, au sein d'un programme, des fragments de code quasiment identiques. Cependant, la répétition était encore plus présente entre les programmes (différentes programmes partagent plusieurs fonctionnalités). Pour pallier à ce problème, on a inventé un style de programmation qui s'appelle la *programmation modulaire*. Cette dernière consiste à décomposer une grosse application en modules, groupes de

fonctions, de méthode et de traitement, pour pouvoir les développer indépendamment et les réutiliser dans d'autres applications.

Les avantages d'une programmation modulaire sont multiples :

- Réduction de la taille des programmes sources améliorant la lisibilité et réduisant les temps de compilation,
- Structuration accrue de l'application par la conception de modules fonctionnellement indépendants,
- Le développement du code des modules peut être attribué à des (groupes de) personnes différentes, qui effectuent leurs tests unitaires indépendamment,
- Limite ou supprime les doublons de code identique, ce qui facilite la maintenance du programme (un bug est corrigé une seule fois),
- Facilite grandement la réutilisabilité et le partage de code.

3. Les Objets et les Classes

a) Notion d'objet : Un objet représente une entité du monde réel, ou de monde virtuel dans le cas d'objets immatériels, qui se caractérise pas une identité, des états significatifs et par un comportement. Un programme est vu comme une société d'objets. Au cours de son exécution, ces objets collaborent en s'envoient des messages dans un but commun.

Objet = Identité + Etat + Comportement

L'identité constitue le moyen de distinguer un objet par rapport aux autres objets d'un système.

L'état définit l'une des possibilités dans laquelle peut se trouver un objet à un moment donné de sa vie.

Le comportement définit la manière dont l'objet agit ou réagit aux divers messages qui lui parviennent de son environnement.

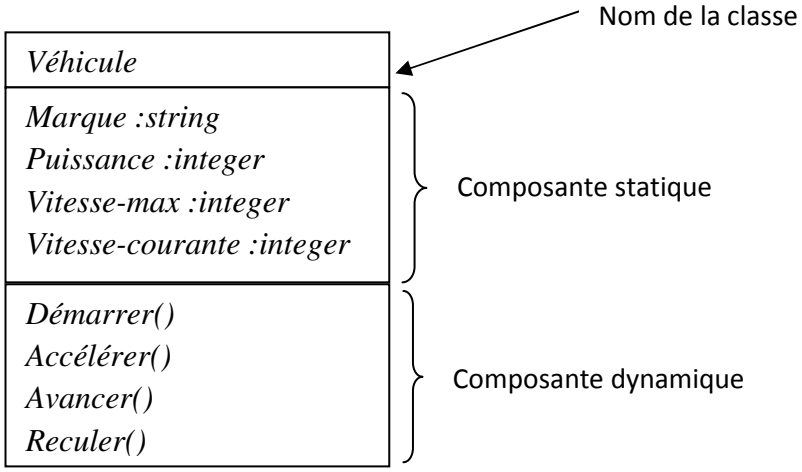
b) Notion de classe : une classe correspond simplement à la généralisation de la notion de type que l'on rencontre dans les langages classiques. En effet, une classe n'est rien d'autre que la description d'un ensemble d'objets ayant une structure de données commune et disposant des mêmes méthodes. Les objets apparaissent alors comme des variables d'un tel type classe (en P.O.O., on dit aussi qu'un objet est une "instance" de sa classe).

Une classe décrit un modèle d'objets ayant tous la même structure et le même comportement. Une classe peut être considérée comme un moule à partir duquel on peut créer des objets. Elle

décrit la structure interne d'un objet, les données qu'il regroupe et les actions qu'il est capable d'assurer sur ces données. Chaque classe se caractérise par deux composantes :

a) *Composante statique* : une liste de champs nommés *données* qui caractérisent l'état des objets de la classe pendant l'exécution.
b) *Composante dynamique* : les méthodes qui manipulent les champs. Elles caractérisent le comportement commun des objets de la classe : ce sont les actions que l'on peut appliquer à chaque objet de la classe.

Exemple : la classe *Véhicule*



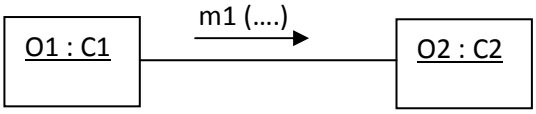
c) Attributs : Un attribut est une information élémentaire qui caractérise une classe et dont la valeur dépend de l'objet instancié. Le type d'un attribut défini a priori.

Exemple : Dans la classe Véhicule, Marque, Puissance, Vitesse-max et Vitesse-courante sont des attributs.

d) Notion de message :

Un objet seul ne permet pas de concevoir une application garantissant les objectifs de la Programmation orientée objet .Un programme est constitué d'objets. Ces derniers communiquent à l'aide de messages. Un message est composé : du nom de l'objet recevant le message, du nom de la méthode à exécuter et des paramètres nécessaires à la méthode.

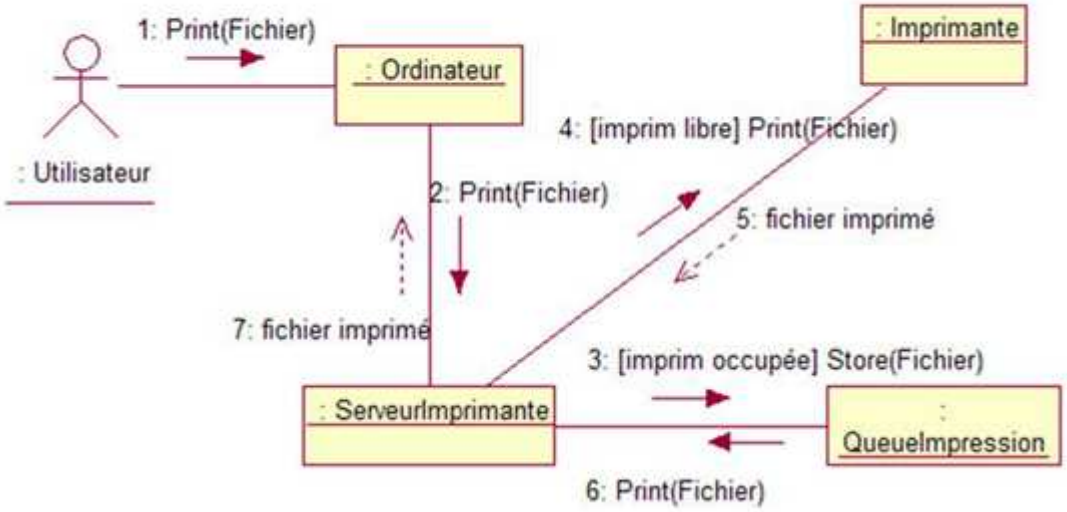
Exemple : La figure suivante montre un échange de message entre l'objet O1 de la classe C1 et l'objet O2 de la classe C2. L'objet O1 envoie le message m1(…) à l'objet O2. Dès qu'il aura reçu ce message, l'objet O2 exécute la méthode m1(…) pour en répondre.



e) Résolution de problème par échange de messages

Pour résoudre un problème, les objets d'un programme orientée objet doivent collaborer en s'échangeant des messages.

Exemple : La figure suivante montre les échanges de messages entre les différents objets d'un système informatique pour imprimer un fichier. Les messages sont numérotés pour montrer leur ordre d'exécution.



4. Introduction à JAVA

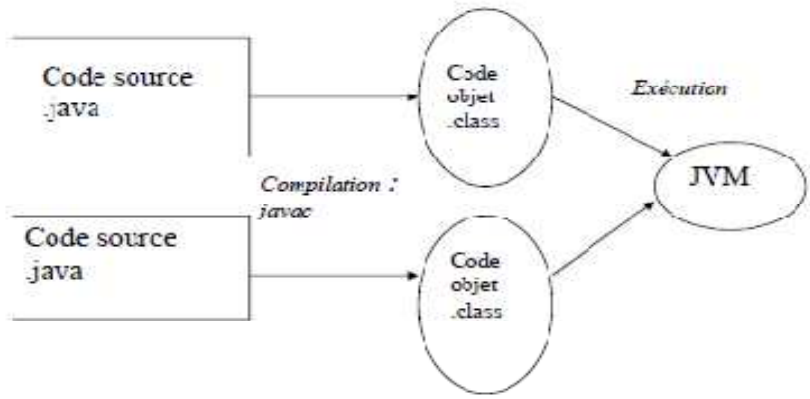
Java est un langage de programmation d'applications orientées objet conçu et développé par les ingénieurs de la société Sun Microsystems en 1995. Cette société a été ensuite rachetée par ORACLE Corporation en Avril 2009 .

D'après la société Sun Microsystems : "*Java est un langage simple, orienté objet, distribué, robuste, sûr, indépendant des architectures matérielles, portable, de haute performance, multithread et dynamique*".

Il existe plusieurs plateformes Java, nous citons entre autres : Java SE (Standard Edition), Java ME (Micro Edition) et Java EE (Entreprise Edition).

- **JVM (Java Virtual Machine)**

JVM est une machine abstraite. C'est ce qu'on appelle une machine virtuelle car elle n'existe pas physiquement. C'est une spécification qui fournit un environnement d'exécution dans lequel le bytecode Java peut être exécuté. La machine virtuelle interprète le bytecode : décode une instruction de bytecode à la fois et exécute l'opération correspondante.



- **JRE (Java Runtime Environment)**

L'environnement d'exécution Java (abr. JRE pour Java Runtime Environment), permet l'exécution des programmes écrits en langage de programmation Java, sur différentes plateformes informatiques.

Le JRE se compose d'une machine virtuelle, de bibliothèques logicielles utilisées par les programmes Java

- **JDK (Java Development Kit)**

Le kit de développement Java (JDK) est un environnement de développement logiciel utilisé pour développer des applications et des applets Java. Cela existe physiquement. Il contient JRE + development tools.

JDK est une implémentation de l'une des plates-formes Java (Java SE, Java EE, Java ME). Le JDK contient une machine virtuelle Java privée (JVM) et quelques autres ressources telles qu'un interpréteur / loader (java), un compilateur (javac), un archiveur (jar), un générateur de documentation (Javadoc), etc.

JRE = JVM + Java Class Library (JCL)
JDK = JRE + Java Development Tools (JDT)

a) Types et structures de contrôle en JAVA

- **Types de données primitifs** : La table suivante montre les huit types de données primitifs en Java.

Type de données	Taille	Description
byte	1 byte	Stocke des nombres entiers de : -128 à 127
short	2 bytes	Stocke des nombres entiers de : -32768 à 32767
int	4 bytes	Stocke des nombres entiers de : -2147483648 à 2147483647
long	8 bytes	Stocke des nombres entiers de: -9223372036854775808 à 9223372036854775807
float	4 bytes	Stocke les nombres fractionnaires. Suffisant pour stocker 6 à 7 chiffres décimaux
double	8 bytes	Stocke les nombres fractionnaires. Suffisant pour stocker 15 chiffres décimaux
char	2 bytes	Stocke un seul caractère
boolean	1 bit	Stocke les valeurs vrai ou faux

- **Structures de contrôle en JAVA**

La table suivant montre les quatre instructions conditionnelles en Java.

Instruction	Description
if	Spécifie un bloc de code à exécuter, si une condition spécifiée est vraie
else	spécifie un bloc de code à exécuter, si la même condition est fausse
else if	Spécifie une nouvelle condition à tester, si la première condition est fausse
switch	spécifie plusieurs blocs de code alternatifs à exécuter

Exemple 1: l'instruction if, else if, else

```
if (x > 0) {
            System.out.print(" le nombre est positif");
            }
else if  (x < 0) {
            System.out.print(" le nombre est négatif");
                  }
     else {
            System.out.print(" le nombre est nul") ;
           };
```

Exemple 2: l'instruction switch

```
switch (day) {
  case 1:  System.out.println("Dimanche");  break;
  case 2:  System.out.println("Lundi");  break;
  case 3:  System.out.println("Mardi"); break;
  case 4:  System.out.println("Mercredi");   break;
  case 5:  System.out.println("Jeudi");   break;
  case 6:  System.out.println("Vendredi");  break;
  case 7:  System.out.println("Samedi");   break;
  default: System.out.println("Erreur");   break;

}
```

b) Classe et instanciation

Une classe est un ensemble de données et de fonctions regroupées dans une même entité. L'instanciation de classes est un concept clé de la programmation objet java. Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable. Pour créer un objet en Java on utilise souvent l'opérateur **new**.

Syntaxe d'instanciation :

nom_de_classe **nom_d_objet** = **new** nom_de_classe(...);

Exemple :

```
public class point {

    int x;
    int y;
    public void assigner (int x1, int y1)
    {
            x=x1;
            y=y1;
    }
    public void afficher()
    {
            System.out.println(x);
            System.out.println(y);
    }
    }

   /*****************************classe principale**********************/

     public class principal {

    public static void main(String[] args) {
     //creation d'un objet (instance) de type point
                point p=new point();
     // appel de la fonction assigner()
                p.assigner(10, 20);
     // appel de la fonction afficher()
                p.afficher();

    }
}
```

Résultat d'exécution :

```
10
20
```

c) Méthodes

En Java, les fonctions s'appellent des méthodes.La notion de méthodes en Java est semblable à celle de fonctions en langage C. Une méthode comporte **une en-tête** avec une **liste d'arguments** (paramètres formels) et un **corps** qui contient **les instructions de la méthode** comme l'indique la syntaxe ci-dessous :

```
<type du résultat><nom de méthode> (type1 argument1, type2  argument2, ...)  //en-tête
                                                                              //corps
    {

      //instructions

    }
```

- **type_du résultat** représente le type de valeur que la méthode est sensée retourner (char, int, float,...).
- Si la méthode ne renvoie aucune valeur, on la fait alors précéder du mot-clé ***void.***
- Une méthode doit obligatoirement porter un type de retour (sauf dans le cas des constructeurs)
- le **nom de la méthode** suit les mêmes règles que les noms de variables :
  - le nom doit commencer par une lettre
  - un nom de méthode peut comporter des lettres, des chiffres et les caractères _ et $ (les espaces ne sont pas autorisés!)
  - le nom de la méthode, comme celui des variables est sensible à la casse (différenciation entre les minuscules et majuscules)
- Les **arguments** sont facultatifs, mais s'il n'y a pas d'arguments, les parenthèses doivent rester présentes.
- Les méthodes en Java peuvent renvoyer un résultat de n'importe quel type élémentaire ou objet grâce au mot clef **return.**
- En java les méthodes (fonctions) ne peuvent se trouver qu'à l'intérieur de classes.

Exemple : addition de deux nombres entiers.

```
int addition (int x, int y)
{
    return x+y ;
}
```

d) Les références et passage de paramètres

Il existe deux modes de passage de paramètre en Java, à savoir, le passage par valeur et le passage par référence.

- **Passage par valeur :** ce type de passage est réservé uniquement aux types primitifs (int, byte, short, long, boolean, double, float, char).

  Exemple :

```
public class PassageParValeur {

        public static void ModifierValeur(int y)
        {
              System.out.println("Y= " + y +  "  dans la méthode ModifierValeur");
               y=y+10;
              System.out.println("Y= " + y +  "  dans la méthode ModifierValeur");
        }

        public static void main(String[] args)
        {
             int x=10;
             System.out.println("X= " + x +  "  avant l'appel");
              ModifierValeur(x);
             System.out.println("X= " + x +  "  après l'appel");
        }
  }
```

Résultat d'exécution :

```
X= 10  avant l'appel
Y= 10  dans la méthode ModifierValeur
Y= 20  dans la méthode ModifierValeur
X= 10  après l'appel
```

Nous constatons que la méthode *ModifierValeur* a modifié la valeur du paramètre formel Y et la valeur du paramètre effectif X reste inchangé.

Règle : Le passage par valeur ne permet pas de modifier, dans la méthode appelante, le contenu de la variable passée en paramètre (paramètre effectif).

- **Passage par référence :** ce type de passage est réservé uniquement aux types objets.

  Exemple :

```
public class Point {
   int abscisse;
   int ordonnée;
public Point(int x, int y)      // constructeur
   {
         abscisse=x;
         ordonnée=y;
   }
}
```

```
public class PassageParRéférence {

public static void DeplacerPoint(Point Q, int x, int y)
  {
      System.out.println("Q.abscisee = " + Q.abscisse +  "  dans la méthode DepacerPoint");
      System.out.println("Q.ordonnée = " + Q.ordonnée +  "  dans la méthode DepacerPoint");
     Q.abscisse=Q.abscisse + x;
     Q.ordonnée=Q.ordonnée + y;
     System.out.println("Q.abscisee = " + Q.abscisse +  "  dans la méthode DepacerPoint");
     System.out.println("Q.ordonnée = " + Q.ordonnée +  "  dans la méthode DepacerPoint");
  }

public static void main(String[] args) {
    Point P= new Point(0,0); // creation d'un objet avec abscisse==0 et  ordonnée==0
    System.out.println("P.abscisee = " + P.abscisse +  "  avant l'appel  de la méthode
                                                        DepacerPoint");
    System.out.println("P.ordonnée = " + P.ordonnée +  "  avant l'appel  de la méthodee
                                                        DepacerPoint");
    DeplacerPoint (P, 10, 20);      // l'appel
    System.out.println("P.abscisee = " + P.abscisse +  "  après l'appel  de la méthodee
                                                        DepacerPoint");
    System.out.println("P.ordonnée = " + P.ordonnée +  "  après l'appel de la méthodee
                                                        DepacerPoint");
}
```

Résultat d'exécution :

P.abscisee = **0** avant l'appel de la méthodee DepacerPoint
P.ordonnée = **0** avant l'appel de la méthodee DepacerPoint
Q.abscisee = 0 dans la méthode DepacerPoint
Q.ordonnée = 0 dans la méthode DepacerPoint
Q.abscisee = 10 dans la méthode DepacerPoint
Q.ordonnée = 20 dans la méthode DepacerPoint
P.abscisee = **10** après l'appel de la méthodee DepacerPoint
P.ordonnée = **20** après l'appel de la méthodee DepacerPoint

Nous constatons que les valeurs des variables de l'instance (objet) P ont changé. Cela veut dire que les modifications apportées sur l'objet passé en paramètre à l'intérieur de la méthode sont visible en dehors de la méthode.

Règle : Le passage par référence permet de modifier, dans la méthode appelante, le contenu de l'objet passé en paramètre (paramètre effectif).

Remarque : Un paramètre effectif peut être une constante, une variable, une expression ou une autre méthode retournant une valeur de type compatible avec le type du paramètre formel correspondant.