INTELLIGENTSIA CORPORATION

CENTRE NATIONAL D'ORIENTATION ET DE PRÉPARATION AUX CONCOURS D'ENTRÉE DANS LES GRANDES ÉCOLES ET FACULTÉS DU CAMEROUN

SINCE 2006

CENTRE NATIONAL D'ORIENTATION ET DE PREPARATION AUX CONCOURS D'ENTREE DANS LES GRANDES ECOLES ET FACULTES DU CAMEROUN

Préparation au Concours d'Entrée en Troisième Année de l'ENSP et FGI

Support de Cours

PROGRAMMATION ORIENTÉE OBJET : JAVA

Avec Intelligentsia Corporation, Il suffit d'y croire !!!

☎ 698 222 277 / 671 839 797
fb : Intelligentsia Corporation
email : contact@intelligentsia-corporation.com

MARS 2019

" Vous n'êtes pas un passager sur le train de la vie, vous êtes l'ingénieur. "

-- Elly Roselle --

Instructions :

Il est recommandé à chaque étudiant de traiter les exercices de ce recueil (du moins ceux concernés par la séance) avant chaque séance car ***le temps ne joue pas en notre faveur.***

I. INTRODUCTION & OBJECTIFS

La **programmation orientée objet** (**POO**), ou programmation par objet, est un paradigme de programmation informatique élaboré au début des années 1960 et poursuivi par les travaux de l'Américain Alan Kay dans les années 1970. Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

Java est un langage de programmation orienté objet créé par Sun Microsystems dans les années 95 et racheté par Oracle en 2009.

La particularité et l'**objectif central de Java** est que les logiciels écrits dans ce langage doivent être très facilement portables sur plusieurs systèmes d'exploitation tels que Unix, Windows, Mac OS ou GNU/Linux, avec peu ou pas de modifications.

NB : Pour bien comprendre ce cours, il est important d'avoir suivi un cours sur la programmation en C (ou tout autre langage de syntaxe similaire) car ici on ne reviendra pas sur les notions élémentaires telles que : Les types primitifs, les types de retour des fonctions, les tests, les boucles, les opérations arithmétique etc.

Le cours est essentiellement basé sur les concepts OO (Orienté Objet) propres au langage Java.

II. CARACTERISTIQUES DE JAVA

Java est interprété	le code source est compilé en pseudo code ou byte code puis exécuté par un interpréteur Java : la Java Virtual Machine (JVM). Ce concept est à la base du slogan de Sun pour Java : WORA (Write Once, Run Anywhere : écrire une fois, exécuter partout). En effet, le byte code, s'il ne contient pas de code spécifique à une plate-forme particulière peut être exécuté et obtenir quasiment les même résultats sur toutes les machines disposant d'une JVM.
Java est portable : il est indépendant de toute plate-forme	il n'y a pas de compilation spécifique pour chaque plateforme. Le code reste indépendant de la machine sur laquelle il s'exécute. Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une Java Virtual Machine. Cette indépendance est assurée au niveau du code source grâce à Unicode et au niveau du byte code.
Java est intuitif	La syntaxe du Java est très proche de celle de ses prédécesseurs (C/C++). Les tests, les boucles, les opérations, les entrées-sorties etc ont été repris exactement comme en C/C++.
Java est orienté objet.	comme la plupart des langages récents, Java est orienté objet. Chaque fichier source contient la définition d'une ou plusieurs classes qui sont utilisées les unes avec les autres pour former une application. Java n'est pas complètement objet car il définit des types primitifs (entier, caractère, flottant, booléen,...).
Java est simple	le choix de ses auteurs a été d'abandonner des éléments mal compris ou mal exploités des autres langages tels que la notion de pointeurs (pour éviter les incidents en manipulant directement la mémoire), l'héritage multiple et la surcharge des opérateurs, ...
Java est fortement typé	toutes les variables sont typées et il n'existe pas de conversion automatique qui risquerait une perte de données. Si une telle conversion doit être réalisée, le développeur doit obligatoirement utiliser un cast ou une méthode statique fournie en standard pour la réaliser.
Java assure la gestion de la mémoire	l'allocation de la mémoire pour un objet est automatique à sa création et Java récupère automatiquement la mémoire inutilisée grâce au garbage collector (ramasse-miettes) qui restitue les zones de mémoire laissées libres suite à la destruction des objets.
Java est sûre	la sécurité fait partie intégrante du système d'exécution et du compilateur. Un programme Java planté ne menace pas le

	système d'exploitation. Il ne peut pas y avoir d'accès direct à la mémoire. L'accès au disque dur est réglementé dans une applet.
Java est multitâche	il permet l'utilisation de threads qui sont des unités d'exécution isolées. La JVM, elle-même, utilise plusieurs threads.
Java est économe	le pseudo-code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution.

III. LES GRANDS CONCEPTS DE LA POO

Un logiciel informatique peut être vu comme un ensemble d'objets (ou entités) contenant des traitements internes (méthodes) ou d'autres données (attributs) et interagissant entre eux. L'approche orienté objet consiste donc à identifier ces différents objets et leurs interactions.

Il existe 4 concepts fondamentaux sur lesquels repose la POO :

1. L'encapsulation

C'est le fait de regrouper au sein d'une même entité les données et les traitements : On parle généralement de classe.

Une **classe** est l'abstraction d'un ensemble d'objets qui possèdent une structure identique (même liste d'attributs) et un même comportement (même liste de méthodes).

Un **objet** est donc une instance d'une classe.

a) La classe

En JAVA, Une classe comporte sa déclaration, des variables et la définition de ses méthodes.

La classe comprend deux parties : un en-tête et un corps.

Le corps peut être divisé en 2 sections :

- La déclaration des données et des constantes
- La définition des méthodes.

Les méthodes et les données sont pourvues d'attributs de visibilité qui gère leur accessibilité par les composants hors de la classe.

La syntaxe de déclaration d'une classe est la suivante :

```
modificateur class nom_de_classe [extends classe_mere] [implements interfaces]
{
...
}
```

Les modificateurs de classe (ClassModifiers) sont :

Modificateur	Rôle
abstract	(voir abstraction).
final	la classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes enfants (voir héritage).
private	la classe n'est accessible qu'à partir du fichier où elle est définie
public	La classe est accessible partout

NB : Les modificateurs **abstract** et **final** ainsi que **public** et **private** sont mutuellement exclusifs.

- Les données d'une classe sont contenues dans des variables appelées propriétés ou attributs. Ce sont des variables qui peuvent être :
  - ✓ Des variables d'instances nécessitent simplement une déclaration de la variable dans le corps de la classe ;
  - ✓ Des variables de classes communes à toutes les instances de la classe et définies avec le mot clé **static**.

    Ex : `static int compteur ;`
  - ✓ Des constantes (leur valeur ne peut pas être modifiée une fois qu'elles sont initialisées) définies avec le mot clé **final**.

    Ex : `final double PI = 3.14`

- Les méthodes sont des fonctions qui implémentent les traitements de la classe. La syntaxe de la déclaration d'une méthode est :

```
modificateurs type_retourné nom_méthode ([type_arg1 nom_arg1, ... ])
{
...    //définition des variables locales et du bloc d'instructions
}
```

Les modificateurs d'accès des méthodes sont :

Modificateur	Rôle
public	la méthode est accessible aux méthodes des autres classes
private	l'usage de la méthode est réservé aux autres méthodes de la même classe
protected	la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous classes
final	la méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
static	la méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	la méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
native	le code source de la méthode est écrit dans un autre langage

NB : Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package dans lequel la classe se situe.

Remarque : Il existe des méthodes particulières telles que :

- Le constructeur :
  Il permet de faire l'initialisation des attributs de la classe lors de son instanciation. Le constructeur suit la définition des autres méthodes excepté que son nom qui doit obligatoirement correspondre à celui de la classe et qu'il n'est pas typé (pas même **void**), donc il ne peut pas y avoir d'instruction return dans un constructeur. On peut surcharger un constructeur (voir polymorphisme). La définition d'un constructeur est facultative. Si elle n'est pas définie, la machine virtuelle appelle un constructeur par défaut vide créé automatiquement. Dès qu'un constructeur est explicitement défini, Java considère que le programmeur prend en charge la création des constructeurs et que le mécanisme par défaut, qui correspond à un constructeur sans paramètres, est supprimé. Si on souhaite maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètres.

  Ex :

```
public class MaClasse
{
      private   int nombre ;
      public MaClasse()
      {
            nombre = 0 ;
      }
}
```

- **Les accesseurs (getters) et les mutateurs (setters)**:

  L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées **private** à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés « échanges de message ». Un accesseur est une méthode publique qui donne l'accès à une variable d'instance privée.
  Par convention, les accesseurs en lecture commencent par **get** et les accesseurs en écriture commencent par **set**.
  NB : Les accesseurs en écriture sont aussi appelés mutateurs.

  Ex : Pour la classe MaClasse précédente, on peut avoir :

```
public int getNombre()
{
      return nombre ;
}
public void setNombre(int nombre)
{
      this.nombre = nombre ;
}
```

Remarques :

- Le mot clé **this** est utilisé pour faire référence à l'objet courant lorsqu'on est à l'intérieur d'une classe. Il permet aussi souvent de lever l'ambigüité lorsqu'on a des noms de paramètres qui coïncident avec des noms d'attributs comme dans le « setter » précédent.
- **this()** fait référence au constructeur.

b) Les objets

Les objets possèdent des attributs et des méthodes. Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet.

En Java, une application est un objet. La classe est la description d'un objet. Un objet est une instance d'une classe. Pour chaque instance d'une classe, le code est le même, seules les données sont différentes à chaque objet.

Pour déclarer un objet Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré. La déclaration est de la forme

```
nom_de_classe nom_de_variable;
```

Exemple :

```
MaClasse m;
```

Ensuite l'opérateur **new** se charge de créer une instance de la classe et de l'associer à la variable.

Exemple :

```
m = new MaClasse();
```

Il est possible de tout réunir en une seule déclaration

Exemple :

```
MaClasse m = new MaClasse() ;
```

Une fois un objet instancié, l'accès aux méthodes et aux attributs (qui sont déclarés **public**) se font par le nom de l'objet suivit d'un point (« . ») puis du nom de la variable ou méthode correspondante.

Exemple :

```
int nombre = m.getNombre() ;
```

2. L'héritage

Il permet de construire de nouvelles classes (classes enfants) à partir d'autres classes (classes parentes). Ici la classe fille héritera des attributs et des méthodes de la classe mère. L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes.

En JAVA, Le mot clé **extends** permet de spécifier une superclasse éventuelle (ou classe parente) : ce mot clé permet de préciser la classe mère dans une relation d'héritage.

Pour faire appel au constructeur de la classe mère on utilise l'instruction « **super**() ; ».

pareillement pour avoir accès à une méthode de la classe mère on utilise la syntaxe suivante : super.nom_méthode(liste_arguments) ;

Exemple :

```
public class Ville
{
      protected String nom ;
      protected int nbreHabitants ;

      Ville() {  nbreHabitants = 0 ; }

      Ville(String n, int nbrehabitant)
      {
            nbreHabitants = nbrehabitant ;
      }
}
```

```
public class Capitale extends Ville
{
      Capitale()
      {
            super();
      }

      Capitale(String n, int nbreHabitant)
      {
            super(n, nbreHabitant);
      }
}
```

NB : l'héritage multiple n'est pas possible en java mais peut être simulé avec des interfaces.

3. Le polymorphisme

C'est est la capacité donnée à une même opération de s'exécuter différemment suivant le contexte de la classe où elle se trouve. Ainsi une opération définie dans une super-classe peut s'exécuter de manière différente selon la sous-classe où elle est héritée.

On en distingue deux types en POO :

a) La surchage (overloading en anglais) :

En JAVA, la surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des arguments. Ceci permet de simplifier l'interface des classes vis à vis des autres classes.

Une méthode est surchargée lorsqu'elle exécute des actions différentes selon le type et le nombre de paramètres transmis. Il est donc possible de donner le même nom à deux méthodes différentes à condition que les signatures de ces deux méthodes soient différentes. La signature d'une méthode comprend le nom de la classe, le nom de la méthode et les types des paramètres.

Exemple :

```
class Affiche
{
	public static void afficheValeur(int i)
	{
		System.out.println("Nombre entier: " + i);
	}

	public static void afficheValeur(float f)
	{
		System.out.println("Nombre flottant: " + f);
	}
}
```

b) La redéfinition ou masquage (overriding en anglais) :

Il s'agit lors de l'héritage de redéfinir une méthode préalablement définie dans la classe mère ; Celle de la classe mère sera en quelque sorte masquée lorsqu'il s'agira d'un objet instancié à partir du constructeur la classe fille.

Exemple :

```
class A
{
	public void affiche()
	{
		System.out.println("Je suis un objet de type A");
	}
}
class B extends A
{
	@Override
	public void affiche()
	{
		System.out.println("Je suis un objet de type B");
	}
}
```

Remarque :

- Lorsqu'il s'agit d'une redéfinition, l'annotation « **@Override** » est conseillée pour la clarté du code (mais pas obligatoire).
- Une méthode marquée **final** ne peut pas être redéfinie.

4. L'abstraction

L'abstraction consiste à définir des modèles de données en décrivant sommairement les attributs et les méthodes de ces modèles. Tels quels ils ne sont pas utilisables mais ces derniers peuvent servir de moule pour construire des solutions à des problèmes concrets.

En JAVA, une classe avec pour modificateur **abstract** contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstraite ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.

Exemple :

```
abstract class Forme
{
	public abstract float perimetre();
}
class Cercle extends Forme
{
	private float rayon;

	Cercle(float rayon)
	{
		this.rayon = rayon;
	}
	@Override
	public float perimetre()
	{
		return 2*Math.PI*rayon;
	}
}
```

Les interfaces :

En java une interface est comme une classe abstraite, avec les différences suivantes :

- ✓ Au lieu d'héritage, on parle d'implémentation
- ✓ Une classe peut implémenter plusieurs interfaces
- ✓ Une interface ne contient que des définitions de méthodes, il n'est pas possible d'y ajouter des attributs ou d'implémenter une méthode.

Exemple :

```
public interface Affichable
{
	public void afficher();
}
public interface Redimentionnable
{
	public void doubler();
}
class Carre extends Forme implements Affichable, Redimentionnable
{
	private float cote;

	Carre(float cote)
	{
		this.cote = cote;
	}

	@Override public float perimetre() { return 4*cote; }

	public void afficher()
	{
		System.out.println("Carré de côté " + cote);
	}

	public void doubler() { cote *= 2; }
}
```