**Document pour préparation au test informatique**

Pour être admis en LOG121, vous devez au préalable avoir une compréhension des concepts suivants[[1]](#footnote-1) :

* Ce qu'est un **attribut**
* Ce qu'est une **méthode**
* Ce qu'est une **classe**
* Ce qu'est un **objet** ou une **instance**
* Ce qu'est l'**encapsulation**
* Ce qu'est la **surcharge**
* Ce qu'est la **redéfinition**
* Ce qu**’est la référence this**
* Ce qu'est l'**héritage** par extension
* Ce qu**’est la référence super**
* Ce qu'est l'implémentation d'**interface**
* Ce qu'est le **polymorphisme**
* Ce qu'est la **gestion événementielle**
* Ce que sont les composants **SWING** et la construction d'écran utilisateur graphique de style **GUI (Graphical User Interface)**
* Ce qu'est le **chaînage** **dynamique**
* Ce qu’est **la création, la levée et la gestion d’exceptions.**
* Ce qu'est un **type de données abstrait** (TDA)
  + - * Ce qu'est une pile (statique et dynamique)
      * Ce qu'est une file        "        "        "
      * Ce qu'est une liste      "        "        "

**Attribut :** Espace mémoire permettant de conserver de l'information pour une catégorie d'objets dans une application. C'est l'équivalent d'un champ d'un enregistrement en programmation impérative. Aussi appelé **variable d’instance.**

**Méthode :** Sous-programme (procédure et fonction) associé à une catégorie d'objets dans une application.

**Classe :** L'endroit où sont définis les attributs et les méthodes qui représentent une catégorie d'objets.  C'est l'équivalent d'un module de type en programmation impérative.

**Objet ou instance :** Pour conserver des données pour un objet, il faut le définir et l'instancier à l'aide du mot réservé **new**.  C'est l'équivalent d'initialiser une variable-enregistrement dynamiquement en programmation impérative sauf qu'en plus, il est possible de démarrer les méthodes définies dans la classe à partir de l'objet.

Dans le texte nous emploierons le mot **objet** plutôt qu'**instance** qui est réservé plutôt aux attributs d’une classe.

**Exemple** :

String chaine;                       //définition d'un objet de la classe String

                       chaine = new String("allo");  //Instanciation de l'objet

                      chaine.charAt(0)                  //Appel de la méthode **charAt** de la classe

//String avec **0** en paramètre effectif, pour

//l'objet nommé **chaine.**

**Encapsulation** : Rendre l'utilisation des attributs d'un objet indépendante de leur implémentation (**private**).  En d'autres termes, on utilise des méthodes pour accéder aux attributs d'une classe plutôt que d'y accéder directement.  C'est ce qui justifie, entre autres, l'utilisation d'accesseurs (get) et/ou de mutateurs (set).  Par exemple : getNom, getAge, setNom, setAge, ...

**La référence this**

Il existe une référence spéciale, utilisable dans une méthode non **static** d’une classe, qui permet d’agir sur les attributs de l’objet actuel. Existe sous différentes formes selon le langage.

**Héritage :** En java, nous parlons d'héritage par hiérarchie de classes.  On peut étendre une classe à l'aide du mot réservé **extends**.  On parle alors d'héritage par extension.  Cela permet d'utiliser le code d'une classe sans avoir à le réécrire.  Cette dernière devient la classe parent et on parle de sous-classe pour les classes enfants.  En Java, une classe ne peut hériter que d'une autre classe (pas d'héritage multiple).

**Exemple** :

**public** **class** Point2D

                                    ...

**public** **class** Point3D **extends** Point2D

La classe Point3D hérite de tout le code non privée **(public** ou **protected)** écrit dans la classe Point2D.

**La référence super**

Au même titre que la référence **this**, il existe une référence spéciale qui permet d’agir sur les attributs non privés (**public** et **protected**) de la classe parent dans une hiérarchie de classe.

**Interface :** L'implémentation d'une interface (**implements**), permet de définir des méthodes qui doivent obligatoirement être implémentées par une  classe qui en hérite. Une classe peut implémenter plusieurs interfaces. Cela permet d'assurer une bonne gestion du polymorphisme. Une classe qui implémente une interface est une sous-classe de cette interface.

**Exemple** :

**public** **class** PanneauQuelconque **extends** JPanel **implements** ActionListener,

MouseListener

La classe **PanneauQuelconque** hérite de tout le code non privé **(public** ou **protected)** écrit dans la classe **JPanel** et doit implémenter les méthodes décrites dans les interfaces **ActionListener** et  **MouseListener**.  PanneauQuelconque est une sous‑classe des interfaces au même titre qu'elle est une sous-classe de **JPanel**.

**Polymorphisme** : Un objet d'une classe peut être instancié pendant l'exécution d'une application avec n'importe quelle sous-classe d'une classe parent.  Dès lors, les méthodes démarrées à partir de l'objet dépendent de la classe d'instanciation.

**Exemple** :

p1 = new Point2D(...);   p1.norme();   //ici la méthode norme appelée est

//celle définie dans la classe Point2D

p1 = new Point3D(...);   p1.norme(); //ici la méthode norme appelée est

//celle définie dans la classe Point3D

**Gestion événementielle :** Les systèmes d'exploitation contemporains avisent les applications des différents événements (simple ou double cliquer sur la souris, taper au clavier, fermer une fenêtre, etc.) qui sont produits par l'utilisateur de l'ordinateur.  Plusieurs événements existent  et les langages de programmation permettent de s'abonner à ces événements pour agir selon les besoins de l'application.  En java, les contrôles se font à l'aide d'interfaces que l'on appelle écouteurs **(Listener)**.

**Surcharge :** Il est possible d'avoir plusieurs méthodes qui ont le même nom dans un même bloc de code en autant qu’elles aient un type de retour ou des paramètres qui diffèrent.

**Redéfinition :** Il est possible d'avoir plusieurs méthodes parfaitement identiques dans une hiérarchie de classes.  Par exemple, une méthode **getDistance()** peut exister dans la classe Point2D et aussi dans la sous-classe Point3D.  C'est le type de l'instance lors de la dernière instanciation qui détermine quelle méthode est appelée.

**Composants Swing** : Classes qui permettent de programmer des applications de style graphique (**G**raphical **U**ser **I**nterface) avec des composants représentant les cadres (**JFrame**), les panneaux (**JPanel**), les boutons (**JButton**), les boîtes de texte (**JTextField**), les boîtes combo (**JComboBox**), etc. Également, vous y trouvez les interfaces de type **Listener** telles **ActionListener**, **MouseListener**, **KeyListener**, ...

**Type de données abstrait** (TDA) **:** Définition d’un type structuré ayant un comportement prédéterminé dont l’utilisation est indépendante de l’implémentation (implémentation encapsulée). Sert essentiellement à conserver des données en mémoire. Le choix du TDA dépend de l’accès aux données que nécessite l’application. S’écrit dans une classe telle **Array**, **Vector**, **ArrayList**, **LinkedList,** etc.

**Statique :** Dans le contexte des TDA, le mot statique signifie que la taille est fixée lors de l’instanciation et qu’elle est non modifiable par la suite.

**Dynamique :** Au contraire de statique, la taille du TDA est variable durant l’exécution du programme. On y fait souvent l’utilisation de chaînage dynamique.

**Chaînage dynamique :** Technique de programmation pour créer des TDA dont la taille varie durant l’exécution. Le(s) lien(s) entre les éléments doi(ven)t être maintenu(s) par le programmeur. Il y a plusieurs techniques de chaînage (simple, double, gauche-droit, etc.) et plusieurs TDA les utilisent.

**Pile :** TDA dont l’ajout ou le retrait d’objets se fait toujours à partir du dernier élément entré (Last In First Out).

**File :** TDA dont l’ajout d’objets se fait à la suite du dernier ajouté et le retrait dans l’ordre d’arrivé (First In First Out ou Last In Last Out).

**Liste :** TDA dont l’ajout ou le retrait d’objets se fait selon les besoins à un endroit spécifié. Il nécessite de fournir la position de l’opération. Cela peut différer selon le type de liste choisie (position courante, itérateur ou indexée).

**Création, levée et gestion d’exceptions :** Il est possible d’envoyer des messages d’exceptions (erreurs) d’une méthode à une autre pendant l’exécution du programme. Un message d’exception démarre lorsqu’il est levé. À partir de ce moment, il est transmis de méthodes appelantes en méthodes appelantes, en respectant la pile des appels du OS, jusqu’à ce qu’elle soit gérée ou que le OS le reçoive. À ce moment, la pile des appels est affichée à l’écran avec le texte associé au message d’exception.

Ce qu’il faut savoir en **Java** pour réussir le test :

**Paquetage (package) :** Les classes de **Java** sont disposées dans des paquetages fournis avec le Java Developpement Kit (**JDK**). Ces paquetages sont installés avec le Java Runtime Environnement (**JRE**) qui contient la Java Virtual Machine (**JVM**) qui permet d’exécuter des programme java. Si votre environnement de programmation est bien installé, vous n’avez pas de manœuvres spéciales à effectuer autre que l’importation pour les utiliser.

**Importation :** Pour utiliser une classe se trouvant dans un **paquetage,** on doit l’importer à l’aide du mot réservé **import** en début de fichier. On peut importer toutes les classes d’un paquetage à l’aide d’une \*

**Exemple** :

**import** java.util.Arrays;

**import** java.util.\*;

Il est possible d’utiliser une classe sans l’importer, il faut alors préfixer le nom de la classe avec le nom du paquetage.

**Exemple** :

**java.util.Arrays** tab = new java.util.Arrays();

**Procédure principale:** Nous favorisons une classe dédiée qui ne contient que le programme principal.

**Exemple** :

**public** **class** DemarrerApplication{

**public** **static** **void** main(String [] args ){

**…** …

}

}

**Graphical User Interface :**  Une application **GUI** se crée dans un cadre (**JFrame** ou **JDialog**) et dispose des composants (**JButton**, **JTextField**, **JComboBox**, **JLabel**, …) dans différents panneaux (**JPanel**). Pour accéder à ces classes, il suffit de les importer à partir de **javax.Swing.**

**Disposition des panneaux :** Nous favorisons la disposition de panneaux à l’aide des gestionnaires de disposition (**LayoutManager**). Le **LayoutManager** du panneau principal (**contentPane)** d’un cadre **JFrame** est **BorderLayout** tandis que celui d’un **JPanel** est **FlowLayout** par défaut. Il en existe plusieurs autres dont **BoxLayout, GridLayout, GridBagLayout, CardLayout, SpringLayout, etc.).** Vous pouvez en consulter la liste à <http://docs.oracle.com/javase/tutorial/uiswing/layout/layoutlist.html>

**Création, levée et gestion d’exceptions :** En java, la classe mère de toutes les exceptions est **Exception**. La classe qui lève une exception doit l’annoncer à l’aide de **throws** dans son entête. Une méthode peut lever plusieurs exceptions à l’aide de **throw.** On gère une exception à l’aide d’un bloc **try-catch**. On affiche la pile des appels l’aide de **e**.printStackTrace() où **e** est une instance de la classe **Exception**. Il est possible d’hériter de la classe **Exception** pour créer nos propres classe d’exceptions.

**TDA fournis**

Nous vous fournissons deux TDAs qui seront utilisés lors du test de classement. Une pile avec implémentation dynamique et une file avec implémentation statique. Les deux classes lèvent des messages d’exception. C’est une bonne idée d’écrire un petit programme principal pour tester les méthodes des TDAs.

1. Peu importe le langage de programmation qui a été utilisé. [↑](#footnote-ref-1)