

Survol des Concepts Orientés Objet en JAVA

Package

Un package est un groupe nommé de fichiers Java `.class`, une collection de classes et d'interfaces, rangés dans un seul répertoire. Ils servent à regrouper des classes participant à un même objectif et à définir un namespace en encapsulant¹ les classes et interfaces qui y sont déclarées. Deux classes du même nom peuvent donc exister dans deux packages différents.

Accès

Pour utiliser des méthodes d'un autre package l'instruction « `import <packageName>` » doit être en tête du code source de la classe appelante. Le seul package qui s'importe tout seul est « `java.lang` ». Une classe dans un package a pour « `fully qualified name` » le nom du package suivi du nom de la classe.

Encapsulation

Un package ne montre que les membres `public` des classe `public`. Il montre également les membres `protected` aux héritiers. Mais, de l'extérieur d'un package, on ne peut pas hériter d'une classe non-publique de celui-ci. De l'intérieur, toute méthode à accès à tout sauf le `private` des autres classes.

Classe

Une classe c'est du code source dans un fichier `.java`. Une fois compilé, le bytecode est enregistré dans un fichier `.class`. Elle devient un objet de type `Class` (instance de la classe `Class`) à son chargement en mémoire JVM.

Un fichier source (`.java`) peut contenir plusieurs déclarations de classes. Une seule de ces classes peut être déclarée `public`. Une classe `public` doit obligatoirement être déclarée dans un fichier source (`.java`) qui porte le même nom que la classe principale (en respectant la casse).

Fichier « `.class` »

Le compilateur `Javac` traduit les instructions de votre code source en bytecode, le « langage machine » de la machine virtuelle (JVM). Un fichier `.class` est un stream binaire (en Unicode, UTF-8) qui peut être exécuté sur n'importe quelle plateforme matérielle et système d'exploitation qui accueille une JVM. Ce format permet l'indépendance vis à vis des plateformes (seul la JVM dépend de la plateforme) et une communication des classes à travers le réseau.

Structure du fichier :

- Une structure contenant le nom de la classe, de la superclasse, des interfaces, des champs, des méthodes et de chaque type (référence ou non) connu lors de la compilation.
- `fields[]` : Une table dont chaque élément donne la description complète d'un champ (= attribut)
- `methods[]` : Une table dont chaque élément donne la description complète d'une méthode et son bytecode. La table contient également les descriptions des méthodes d'initialisation de classe et d'instance (anonymes dans votre code source, mais qui ont reçu les noms `<clinit>` et `<init>` à la compilation)

« Static Initializer » (`<clinit>`)

Une classe peut contenir des blocs statiques d'initialisation. Il peut y en avoir plusieurs dans une même classe. Ce sont des blocs de code qui n'ont pas d'arguments ni de valeur retournée. Ils ne doivent jamais lancer d'exception sinon le chargement de la classe échoue (ils doivent gérer les exceptions en interne). Ces blocs s'écrivent :

```
static {  
    // instructions  
}
```

Lors de la compilation, tous ces blocs seront regroupés, suivant leur ordre d'apparition dans le code, pour former la méthode « Static Initializer ». Il n'en existe donc qu'une par classe. Le fichier `.class` contient une référence symbolique `<clinit>` vers cette méthode.

¹ L'encapsulation est le fait de cacher de l'information. Proposer une série de méthodes pour interagir avec un objet mais sans savoir ce qui se cache derrière. L'objet est un peu comme une « boîte noire ».

Elle sera exécutée lors du chargement de la classe pour permettre une initialisation de la classe et de ses liens avec son environnement. Son principal intérêt vient du fait que c'est la seule manière d'exécuter du code lors du chargement d'une classe, tout comme un constructeur permet d'exécuter du code lors de la création d'une instance.

Les constructeurs

En Java, les constructeurs sont similaires à des méthodes mais ce ne sont pas des méthodes. Ils ont plusieurs caractéristiques :

- Un constructeur a une signature et un corps
- Ils n'ont pas de type retourné (même pas void), ils retournent implicitement this.
- On peut leur spécifier une visibilité.
- Ils ne sont pas hérités par les sous-classes.

Remarque : Si la première instruction du constructeur n'est pas un appel explicite au constructeur de la superclasse avec le mot-clé `super`, alors l'instruction `super()` est implicitement ajoutée. Si la superclasse n'a pas de constructeur sans paramètres, cela provoque une erreur de compilation. Exception : si la première ligne du constructeur est `this(...)` pour invoquer un autre constructeur de la même classe, Javac n'ajoute pas l'instruction `super()` et se pose la question au constructeur suivant.


Instance

Une instance est un sac de données contenant 0 à N variables (d'instance) qui sont des valeurs de types primitifs ou des références à d'autres objets. Si la classe-matrice de l'instance a une superclasse « Parent », le sac sera constitué de 3 couches.

1. (Les valeurs des attributs de la classe « Object » :-s'il y en a, ils sont invisibles)
2. Les valeurs des attributs de la classe « Parent ».
3. Les valeurs des attributs de la classe matrice.

L'état (State) d'une instance est l'ensemble des valeurs de son sac .

Une instance peut être de plusieurs types :

- type Object 
- type de sa classe matrice et de ses ancêtres
- les types des interfaces implémentées et de leurs ancêtres.

« Instance Initializer » (<init>)

Une classe peut contenir des blocs anonymes d'initialisation d'instance. Ils doivent respecter les mêmes conventions que les blocs statiques d'initialisation. Ces blocs s'écrivent :

```
{
    // instructions
}
```

Lors de la compilation, tous ces blocs seront également regroupés, suivant leur ordre d'apparition dans le code, pour former la méthode « Instance Initializer ». Il n'en existe donc qu'une par classe. Le fichier .class contient une référence symbolique <init> vers cette méthode.

Elle sera exécutée lors de chaque instanciation de la classe. Elle ne remplace pas un constructeur :

- Un bloc d'initialisation ne peut pas recevoir de paramètres, un constructeur oui.
- Un bloc d'initialisation permet de partager du code entre plusieurs constructeurs de la même classe
- Sans constructeur, pas de création d'instance.

Instance « Immuable »

Un objet est considéré comme immuable si son état ne peut plus changer, une fois construit (`String`, `Integer`...). A contrario, un objet est muable si son état peut être changé après son instanciation (`Date`...). Comme l'on ne peut pas changer l'état d'un objet immuable, il faut en créer un nouveau si on veut le modifier.

Pour qu'une instance soit immuable, il faut que la classe matrice réponde à certains critères :

- La classe doit être `final`.

- Toutes les variables d'instances doivent être `private` et `final` : pour qu'elles ne puissent pas être accédées de l'extérieur ni être modifiées de l'intérieur.
- Les constructeurs doivent initialiser les variables d'instance avec des copies des paramètres : car sinon la classe qui passe le paramètre pourrait modifier celui-ci par la suite car il connaît sa référence.
- Pas de setters, l'instance doit être construite en une seule fois.
- Les getters renvoient des copies des variables d'instance : car si la classe qui recevait la référence vers l'objet réel elle pourrait le modifier.

Instance « Stateless »

Quand il n'y a aucune variable d'instance dans l'instance (dans les classes parent également), l'instance est Stateless, mais elle existe (généralement en un seul exemplaire), a une adresse en mémoire, et l'on peut invoquer des méthodes d'instance sur sa référence.

Elle est donc immuable, donc ThreadSafe (*Voir plus loin*). On l'appelle « Stateless Service Object ». Sa classe matrice est généralement une classe-utilitaire qui propose des services. Son intérêt est qu'on peut invoquer des méthodes d'instance utilitaires (au travers d'une interface) en profitant du Dynamic binding propres aux méthodes d'instance (*Voir plus loin*).

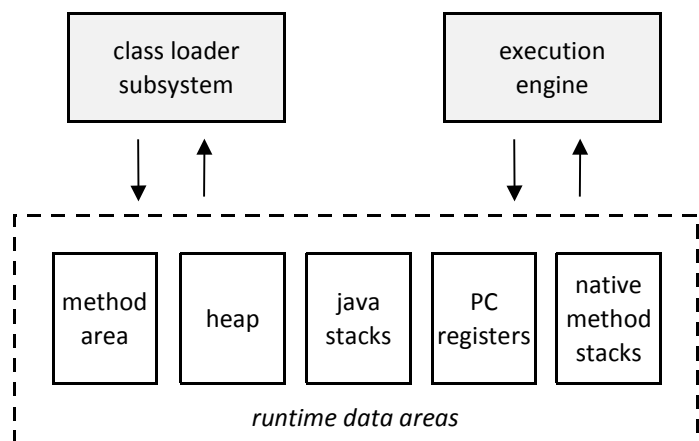
JVM

La JVM est composée de deux systèmes :

- **Un sous-système class loader** : mécanisme de chargement des types (classes et interfaces).
- **Un moteur d'exécution** : mécanisme responsable de l'exécution des instructions contenues dans les méthodes des classes chargées.

Elle est composée également d'une partie « mémoire » qui est subdivisée :

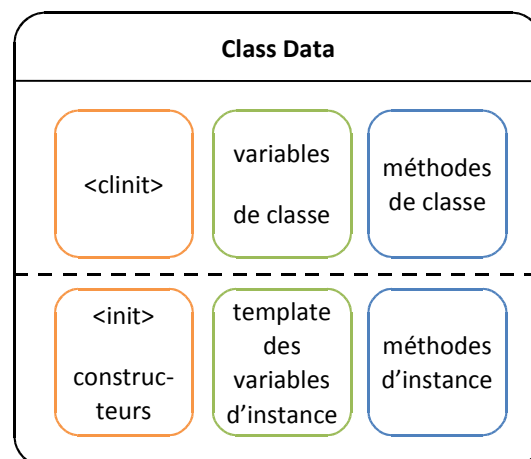
- **La method area** qui contient le code des classes chargées.
- **Le heap** qui contient toutes les instances sous forme de « sacs de bytes ».
- **Les java stacks** qui contiennent un stack par thread en cours d'exécution.
- **Les PC registers**
- **Les native method stacks** qui permettent les appels au système hôte comme des accès aux périphériques...



Chargement d'une classe

Il n'y a pas de classe en mémoire sans chargement. Une classe est chargée dans la JVM quand une méthode d'une autre classe invoque une de ses méthodes. Le chargement est effectué automatiquement par le ClassLoader. Il se déroule comme suit :

-
-
-
-



Tout ce qui concerne la classe en soi, ce qui est static

Tout ce qui concerne les instances de la classe

Exécution du « static Initializer » (<clinit> dans le

Chargement d'une instance

Si le programme rencontre une instruction `new`, ou `newInstance()`, on démarre le processus de création d'une instance :

- Si la classe n'est pas encore chargée en mémoire, on la charge (voir : *chargement d'une classe*)
- La mémoire est allouée dans le Heap pour cette nouvelle instance dont la référence (`this`) est ainsi déjà disponible.
- Ensuite, les « Instance Initializer » et les constructeurs sont exécutés, du haut vers le bas, (classe `Object` en haut, `ClasseMatrice` en bas) dans l'ordre suivant: `{i}`, `(c) Object`, `{i}`, `(c) Ancêtre`, `{i}`, `(c) Parent`, `{i}`, `(c) ClasseMatrice`.
- Enfin, la référence de l'instance est retournée à l'appelant.

« Binding »

Le Binding est le processus qui relie un appel de méthode au bloc d'instructions de la bonne méthode. Certaines liaisons peuvent être faites à la compilation, d'autres doivent être faites au moment de l'exécution par la JVM :

- « **Static binding** » : les appels de méthodes statiques sont liés par le compilateur lors de la compilation.
- « **Dynamic binding** » : les appels de méthodes d'instance sont liés par la JVM lors de l'exécution.

Static binding

Lors de la compilation, le compilateur vérifie si tous les appels de méthodes d'instance peuvent être faits en fonction du type de l'instance. Le type d'une instance définit un sous-ensemble de méthodes disponibles pour cette instance. Mais le compilateur ne peut pas faire le lien avec les méthodes car il ne connaît pas encore les références dans la mémoire de la JVM.

Par contre, les appels de méthodes de classe (méthodes statiques) peuvent déjà être liés car ils ne dépendent pas d'une instance en mémoire.

Dynamic binding

Lors de l'exécution, la JVM peut maintenant faire les liens pour les méthodes d'instance. Elle recherche d'abord la méthode dans la classe-matrice, ensuite en remontant dans l'arbre d'héritage. Elle cherche la méthode la plus « spécialisée » pour l'instance quelque soit son type.

Remarque : Pour un objet de classe-matrice `A` mais de type `Parent`, le compilateur autorisera l'appel de méthodes uniquement définies dans `Parent`. Mais lors de l'exécution, si la méthode a été réécrite dans la classe `A`, c'est celle-là qui sera exécutée et non pas celle de la classe `Parent`.

« Garbage Collector »

Une instance devient inaccessible pour plusieurs raisons : soit la dernière variable qui la référençait est mise à `null`, soit le thread sort du « scope » de cette variable. Dans les deux cas, la place mémoire occupée par cette instance est perdue. Le garbage collector va essayer de libérer ces espaces mémoire. Pour détecter les instances inaccessibles, il y a deux techniques :

- Pour chaque objet dans le heap, la JVM gère un compteur de références. Quand le compteur est 0, l'objet est perdu. Mais cette technique ne détecte pas les cycles de références (plusieurs objets pointent l'un vers l'autre mais son pourtant tous inaccessibles : $A \rightarrow B$, $B \rightarrow A$, mais `A` et `B` ne sont pas référencés ailleurs).
- Le garbage collector parcourt les graphes d'instances à partir des racines. Tous les objets rencontrés dans les graphes sont marqués. En fin de parcours les objets non-marqués sont considérés comme perdus.

Objets²

Types d'objets

Entity

Objets qui ont une identité distincte, qui se modifient dans le temps et qui ont éventuellement plusieurs copies. Ce sont des « gros » objets qui représentent l'activité de l'entreprise. Ils contiennent en général des références à des Value Objects.

² Ici, j'emploierai le terme « objet » dans un sens proche d'un tuple dans une Base de données (Objet Bizz-ness pour le prof) sauf pour les ServiceObjects.

Exemple : Client, Navire, Expédition, Container, Voiture...

Base de données : Ils correspondent en général à une table dans une Base de données.

Value Object

Ce sont des objets inertes qui font partie des données de référence, avec généralement une valeur et une description. Ce sont des « petits » objets qui contiennent des données qui ne changeront pas (ou peu). Ils doivent donc être immuables. Plutôt que de changer une valeur d'attribut, on remplace l'objet complet par un nouveau.

Exemple : CodePostal, CodePays, Devise, Couleur, mais aussi une Adresse (dans les cas simples)...

Base de données : Leur état sera repris dans une table comme « objet imbriqué » dans le tuple de l'Entity qui les référence.

Service Object

C'est un objet utilitaire qui permet d'effectuer des opérations qu'on ne peut pas implémenter dans une autre classe car elles ne concernent pas cette classe directement. Ces opérations sont en général des points de connexion entre plusieurs objets, sans pouvoir être incluse dans un des objets. Cet objet propose donc des services agissant sur plusieurs objets. Il n'a donc en général pas besoin d'attributs et est donc immuable et ThreadSafe.

Exemple : ConnectionBD, une Factory, ... , toutes les INSTANCES captives des interfaces de service.

Identité des objets

MID

Clef temporaire : identifiant dans la JVM, c'est-à-dire sa référence. A chaque exécution, un « même » objet a une MID différente.

Mais ! Tout objet doit avoir un identifiant qui le suive tout au long de sa vie. Qu'il soit dans une Base de données, dans la JVM, dans un fichier XML... Cet identifiant ne doit jamais changer car sinon il faudrait propager le changement dans toutes les copies de l'objet ! Cet identifiant sera l'OID.

OID (ObjectIdentifier)

Clef permanente : une clef artificielle attribuée par le système. Elle n'est pas connue des utilisateurs. Elle doit être mono-attribut, sans signification, unique dans tout l'espace de la société, de même structure pour tous les types d'objets. Cette clef doit également être invariable. Pour cela, il faut qu'elle n'ait pas de signification, car cette signification peut se périmer.

CID (ConceptualIdentifier, Natural-Key)

Clef permanente : l'identifiant d'usage dans l'entreprise. Elle est connue des utilisateurs. Sa valeur peut changer dans le temps et sa structure est différente d'une classe à une autre. Elle peut être également composite (concaténation de plusieurs champs).

Exemples : matricule d'un employé, numéro de plaque d'une voiture, numéro de client, numéro de ligne de facture,...

Le CID n'est pas unique dans l'espace de la société : un numéro de facture peut être le même qu'un numéro de client. Pour rendre cette clef unique, et donc lui donner le rôle de l'OID, il faut l'accompagner de son type d'objet que nous allons dénommer OTY (ObjectType).

Exemple : CLI125 (client 125), CDE54 (commande 54), 12205 (CID*1000+OTY)

Optimistic-Lock

Chaque objet se voit attribuer un numéro de version. Lorsque l'on veut enregistrer l'objet en BD, on vérifie que le numéro de version n'a pas changé (un autre utilisateur aurait pu modifier l'objet entre temps). Si c'est ok, on enregistre l'objet en incrémentant son numéro de version et on fait un commit pour libérer les objets. Sinon, on fait un rollback et on revoie une erreur demandant de recommencer.

Ce système est efficace dans le cas où il est extrêmement rare que deux utilisateurs décident de mettre à jour le même objet au même moment. Sinon, il faudra penser plutôt à un système où l'on bloque l'objet dès le début de l'opération de modification.

Map

Les Map sont des listes de correspondances clef-valeur. Souvent la clef sera le CID ou l'OID et la valeur sera l'objet correspondant. En JAVA, Map est une interface qui est implémentée par HashMap et par AbstractMap. HashMap extends AbstractMap.

```
HashMap<String, Vehicule> plaqueMap = new HashMap<String, Vehicule>(64);

plaqueMap.put(voiture1.plaque, voiture1) // #bucket = voiture1.plaque.hashCode()%64
plaqueMap.get(KPT007)                    // #bucket = "KPT007".hashCode()%64
                                           // En cas de collision, A.equals(B) ?
```

Threads

Un thread est un flux de contrôle dans un programme en cours d'exécution. Un thread parcourt les instructions des méthodes une à une. Chaque instruction parcourue par un thread est interprétée par la JVM.

Pour gérer l'exécution d'un thread, la JVM a besoin de plusieurs choses pour CHAQUE thread :

- Un ensemble de registres. Cet ensemble sera stocké dans la partie « PC Registers » de la JVM
- Un « Stack » pour empiler les « Frames » de chaque méthode ouverte. Ce Stack sera stocké dans la partie « Java Stacks » de la JVM.
- Un « Location Counter ».

Les frames servent à mémoriser l'état de chaque méthode en cours d'exécution (méthode ouverte, méthode courante...). L'état d'une méthode est constitué des valeurs de ses variables locales, la position de la dernière instruction en exécution... Une nouvelle Frame est créée chaque fois qu'une méthode est invoquée, et est détruite lorsque l'invocation se termine. Une frame est allouée sur le Stack du thread qui a créé cette frame.

ThreadSafe

Du code ThreadSafe peut être parcouru par plusieurs threads en parallèle sans « effets de bord », c'est à dire sans problème de concurrence d'accès à des ressources (objets) partagées. Dans une application de gestion, il n'y a que très rarement du multithreading. Cependant, le programme est régulièrement utilisé par plusieurs utilisateurs en même temps sur la même JVM ! Il est donc important de faire du code ThreadSafe.

- Une variable de classe est automatiquement partagée entre tous les threads.
- Une variable d'instance n'est pas partagée entre threads. Sauf si l'instance est connue³ par plusieurs threads.
- Les variables locales sont par définition ThreadSafe car elles sont enregistrées dans une frame et non pas dans le Heap.

Dans un même thread, les objets manipulés sont donc ThreadSafe si ce thread est le seul à connaître l'objet. Cependant, les variables locales (références vers l'objet) sont locales à chaque méthode car elles sont enregistrées dans une Frame ! Il y a donc deux solutions pour partager un objet entre plusieurs méthodes :

- Transmettre l'objet en paramètre à chaque méthode (aie !)
- Utiliser le « pattern Registry ».

Pattern Registry

Le principe du pattern Registry est que chaque thread se voit attribuer une « valise » qui contient les références vers les objets que l'on veut accessibles dans tout le thread. Chaque thread peut avoir une et une seule valise. Cette valise est gardée dans une consigne matérialisée en Java par une `Map<ThreadId, Valise>`. Java propose une classe `ThreadLocal` qui joue le rôle de cette Map.

Chaque thread est une instance de la classe `Thread`. Un thread a un identifiant unique que la JVM nous fournit via la méthode `Thread.currentThread().getId() : long`. Cependant, la classe `ThreadLocal` va chercher automatiquement l'id du thread courant pour lui donner sa Valise. L'utilisation de `ThreadLocal` est donc plus facile par rapport à une implémentation « maison » avec une Map. Voici un exemple d'implémentation d'une consigne à valises :

```
class Valise {           // Les instances ne contiennent que les attributs           = Valise
                        // La classe (static) ne contient que le ThreadLocal         = Consigne

    private static final ThreadLocal<Valise> contextes = new ThreadLocal<Valise>();

    // Attributs d'une Valise
    private MousseARaser mar;
    private Calleecon cal;
    private Singlet sin;

    public static void faisMoiUneValise(){
        // la méthode set() de la classe ThreadLocal range la référence à la Valise
    }
```

³ Connaître une instance c'est posséder, dans une variable locale, la référence de l'instance (accès direct) ou d'une instance plus haut dans le graphe d'instances (accès indirect).

```

        // dans l'entrée de la Map qui correspond au Thread.id du thread courant
        contextes.set(new Valise());
    }
    public static Valise getValise(){
        // la méthode get() de la classe ThreadLocal reprend la référence à la Valise
        // dans l'entrée de la Map qui correspond au Thread.id du thread courant.
        return contextes.get();
    }
}

class Autre {

    void autreMethode() {
        // Récupération de la référence à la Valise pour le thread courant:
        Valise maValise = Valise.getValise();
    }
}

```

Synchronisation

Pour éviter que plusieurs threads exécutent certaines actions sur un même objet (classe ou instance) en même temps, on introduit le concept de synchronisation. Une seule méthode `synchronized` peut être exécutée à la fois sur un même objet, tous threads confondus. On peut parler de « cadenas » sur un objet :

- Tout objet dans le Heap possède un cadenas.
- Lorsqu'on exécute une méthode `synchronized` sur un objet (classe ou instance), on prend le cadenas de l'objet.
- Une méthode non `synchronized` n'a pas besoin de cadenas pour être exécutée.
- Si un thread n'arrive pas à avoir le cadenas d'un objet, le thread attend que le cadenas soit disponible.

Divers

Valeurs par défaut

Java fournit des valeurs initiales par défaut uniquement pour les variables de classe et d'instance : 0 pour un `int`, `false` pour un `boolean`, `null` pour une référence...

Pattern « singleton »

Le pattern singleton propose de n'incarner qu'une seule instance de la classe. C'est à dire qu'il n'y aura en JVM qu'une seule instance de cette classe. Cette instance est partagée entre tous les threads des classes-clientes qui invoquent les méthodes d'instance sur elle.

```

public class Singleton {

    static class SingletonHolder {
        static Singleton instance = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHolder.instance;
    }

    private Singleton(){}
}

```

Abstract = (concerne les instances)

Une méthode `abstract` est une déclaration. L'implémentation est faite dans une autre classe. C'est donc une instruction pour toute sous-classe future : « n'oubliez pas d'implémenter cette méthode ! ».

Remarques :

- Il n'y a pas de méthodes de classe abstraites.
- Il n'y a pas d'attributs abstraits (ni d'instance, ni de classe).
- Il n'y a que des méthodes d'instance qui peuvent être abstraites.

- Une classe abstraite ne peut être « matrice », mais elle peut contribuer à la création d'une instance de sous-classe (une « pelure » de l'oignon)

Introspection

Un objet « portant l'étiquette de sa classe-matrice », il suffit de lui demander cette référence via sa méthode `getClass()`. La classe `Class` propose aussi la méthode `getName()` pour connaître le nom de classe d'un objet descripteur-de-classe.

Si vous connaissez le nom d'une classe, vous pouvez obtenir son objet descripteur en appelant la méthode static `forName()`. Il est possible ensuite de créer d'autres instances de la classe en utilisant la méthode `Class.newInstance()`.

```
Class ocd = Class.forName("LaClasse");  
Object objet1 = ocd.newInstance();           // mais le constructeur par défaut doit être  
                                              // visible dans LaClasse  
                                              // l'objet1 retourné est de type Object
```

Le schéma suivant est très intéressant à comprendre :

http://extranet.ipl.be/OOMADB/document/IPL_objt/Objt01/images/Introspection_survey_m.jpg

Ce qui est important à savoir pour le comprendre, c'est qu'un ODC (Objet Descripteur de Classe) est une instance dans le Heap. On peut donc invoquer des méthodes dessus pour obtenir des informations.

Programmer avec des Interfaces (P2I)

Couplage

Le terme couplage (*coupling*) fait référence à l'interconnexion des classes, la dépendance entre ces classes. On ne peut pas développer une application sans couplage car si il n'y a pas de liens entre les classes, chacune d'elles serait un programme à part entière. Mais l'objectif est un faible degré de couplage.

Il y a deux types de dépendances :

- **La dépendance concrète** : quand une instruction précise l'endroit où se trouve la méthode qui sera exécutée au runtime. C'est-à-dire qu'elle précise la classe qui contient la méthode concrète. La première classe invoque et donc est dépendante d'une implémentation (méthode concrète) d'une autre classe!
- **La dépendance abstraite** : quand une instruction précise le nom d'une méthode abstraite (interface), mais pas la classe où réside la méthode concrète qui sera choisie à l'exécution par la JVM.

Le couplage concret est donc un couplage figé, qui ne changera pas. Un couplage abstrait est un couplage réalisé lors de l'exécution. Il peut donc être modifié facilement (*Voir plus loin*).

Remarque : Une abstraction est une encapsulation d'une implémentation (masque une implémentation) et rend donc votre code indépendant d'une implémentation.

Objectif de la P2I⁴

L'objectif de la P2I est de diminuer les dépendances concrètes et de favoriser les dépendances abstraites par l'utilisation des interfaces.

Un système bien conçu doit maximiser les interactions locales (dans un même package) et minimiser les interactions globales. On tente de concevoir des applications comme un ensemble de packages (des classes *friendly*) qui coopèrent en communiquant via des interfaces bien définies et invariables. Car utiliser une interface c'est éliminer la dépendance concrète vis-à-vis de classes matrices. On la transforme en dépendance abstraite vis-à-vis de l'interface.

Interface

Une interface est une classe dont le code source ne contient que deux parties :

- Les constantes (donc des variables de classe : `static final`)
- Les méthodes d'instance abstraites (sans implémentation)

Une interface est donc :

- **Un contrat de services** : chaque classe qui implémente l'interface doit respecter le « canevas » proposé par l'interface. C'est-à-dire implémenter les méthodes exigées.
- **Un mode d'emploi** : Une classe matrice représente ce qu'une instance est, alors qu'une interface précise ce qu'une instance fait. Il n'y a jamais d'attributs dans une interface (sauf des constantes).

Remarque : On ne change pas le protocole d'une interface. On crée une nouvelle interface qui *extends* la précédente. Ainsi pas besoin de modifier les classes clientes de l'interface initiale.

⁴ Programmation avec des interfaces, « Tout interface ! » (dixit le prof)

Callback

Une interface A peut exiger de la classe cliente⁵ d'implémenter une interface B. Il suffit qu'une méthode de l'interface A ait dans sa signature un objet du type de l'interface B. La classe cliente devra donc implémenter l'interface B pour pouvoir donner sa référence dans la méthode en question.

Exemple :

```
public interface Interface_A {
    public abstract void methode(Interface_B objet);
}

public interface Interface_B {
    public abstract void methodeObligatoire();
}

public class Interface_A_Impl implements Interface_A {
    public void methode(Interface_B objet) {}
}

public class Client implements Interface_B {
    public static void main(String[] args) {
        Interface_A nouvelObjet = (Interface_A) new Interface_A_Impl();
        Client moi = new Client();
        nouvelObjet.methode(moi);
    }

    public void methodeObligatoire() {}
}
```

Factory

Le mot clef `new` est au centre du langage Java. Mais il n'encapsule pas la création d'une instance, et il est non-polymorphe⁶. Si une classe a besoin d'une instance d'une classe dont les particularités lui échappent, ou d'une classe d'un autre package, il vaut mieux faire appel à un « connaisseur » dans le domaine. Une Factory !

La Factory est en quelque sorte le conseiller-livreur des instances.

Exemple : Un package TV contient une interface `ITV` avec les méthodes qu'une TV doit implémenter (allumer, éteindre...). Ce package contient également 17 classes qui implémentent cette interface à leur manière (TVSony, TVSamsung...) et une classe `TVFactory`. Une classe cliente a besoin d'une TV, une instance de `ITV`. Il fait appel à la classe `TVFactory` grâce à la méthode `getInstance()` en précisant ces critères (taille, couleur...). La `TVFactory` renvoie ensuite une instance de `TVSony` (castée en `ITV` car le client s'en fout que c'est une Sony) car c'est celle qui correspond le mieux aux critères.

Si un package contient plusieurs classes (interfaces) fondamentalement différentes (TV, Lecteur DVD...), on peut avoir plusieurs Factories (une par type) ou une seule (pour tous les types). Si une Factory est multi-types, il faudra caster l'instance reçue ou créer une méthode par type.

Le principal avantage d'une Factory est qu'elle permet aux classes clientes d'être indépendantes de la logique d'instanciation : si la classe matrice devient obsolète, il suffit de changer la Factory, les classes clientes ne verront pas la différence.

Interface avec instance captive (SSO)

On pourrait également mettre une interface entre le client et la Factory pour bénéficier du polymorphisme. On va pour cela mettre dans l'interface `TypeFactory` une constante qui sera une référence à un objet de type `ITypeFactoryImpl` qui implémente l'interface. Si l'on veut changer de Factory, il suffira de changer la constante dans l'interface (*Voir plus loin*).

⁵ Classe qui utilise un objet du type de l'interface

⁶ Si l'on utilise une classe XX et que cette classe change, ou est remplacée par une classe XY, il faut changer chaque appel aux constructeurs ! C'est du non-polymorphisme. On ne peut pas utiliser quelque-chose dans un autre contexte.

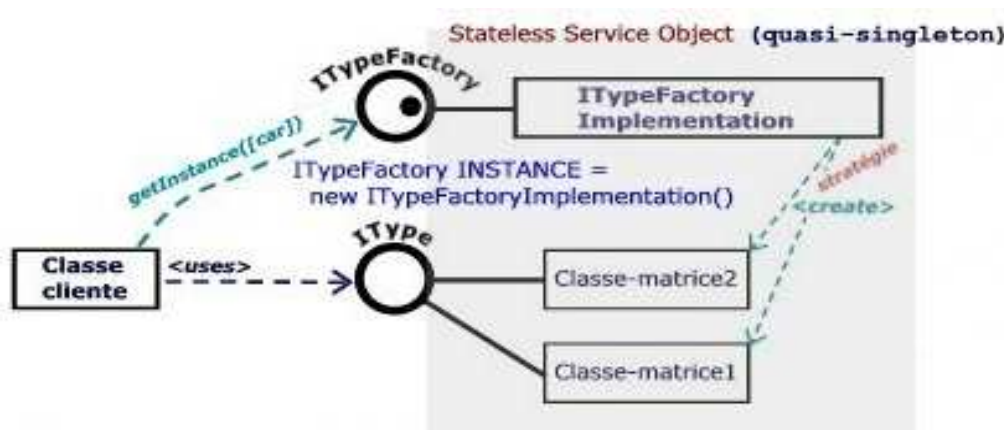
La combinaison de l'interface et de la Factory introduit le concept de SSO (Service Stateless Object).

Interface + Factory = P2I

La P2I regroupe tous les concepts expliqués précédemment. Les classes matrices sont cachées derrière une ou plusieurs interfaces, et le SSO Factory distribue les instances de ces objets. On peut changer les classes matrices sans modifier l'interface, on peut modifier la Factory sans en changer l'interface également. On a supprimé toutes les dépendances concrètes. Le couplage devient minimum, optimal.

Exemple en schéma :

- : représente une interface
- ⊙ : représente une interface avec instance captive (SSO)



Exemple codé :

```
public interface ITypeFactory {
    public abstract Object getInstance();
}

public interface IType {
}

class ITypeFactoryImpl implements ITypeFactory {
    public Object getInstance() {
        return new ClasseMatrice1();
    }
}

class ClasseMatrice1 implements IType {
}

class ClasseMatrice2 implements IType {
}
```

« Program to an interface, not an implementation »

« Show interfaces, not classes »

Rendre les Applications Portables

Rendre une application portable c'est rendre une application indépendante de son environnement, de son contexte d'exécution. Par exemple : changer de DBMS, changer la langue, changer certaines classes d'implémentation...

Il faut pouvoir configurer l'application en plaçant les options de configuration dans un fichier texte qui sera chargé au lancement de l'application.

Properties

Fichier « .properties »

Un fichier .properties est une liste de combinaisons clef – valeur. La clef correspond au nom de la propriété. Elle est suivie du caractère '=' pour signifier la transition entre la clef et la valeur. Ce fichier peut contenir des commentaires : '!' ou '#' en début de ligne.

Exemple :

```
! [DRIVERS] adresse des drivers

#jdbc.drivers = NULL
#jdbc.drivers = com.sybase.jdbc3.jdbc.SybDriver
jdbc.drivers = org.gjt.mm.mysql.Driver
```

Classe « Properties »

La classe Properties permet de gérer ces fichiers de propriétés de manière simplifiée. Elle hérite de la classe Hashtable. Elle contient une liste de propriétés enregistrées dans une Hashtable<String, String>. On peut donc ajouter (setProperty()), récupérer (getProperty()) des propriétés, et également charger des fichiers .properties (load()).

Exemple :

```
private static Properties props = new Properties();

public void loadContext(String nomApplication) {
    FileInputStream fichier = new FileInputStream(nomApplication + ".properties");
    props.load(fichier);
    fichier.close();

    String urlDrivers = props.getProperty("jdbc.drivers");
    . . .
}
```

Contexte

Pour charger le contexte d'une application, nous allons utiliser à nouveau une SSO (Stateless Service Object) qu'on appellera IApplicationContext. Il contiendra une instance de EnvironmentSetup qui va s'occuper du chargement du fichier properties et de l'application des propriétés qui y sont. Elle peut également se charger d'autres opérations telles que charger le Driver JDBC...

Exemple :

```
public interface IApplicationContext {

    public static final EnvironmentSetup INSTANCE = new EnvironmentSetup();

    public abstract void loadContext(String nomApplication);
}

class EnvironmentSetup implements IApplicationContext {

    private static Properties props = new Properties();
```

```

    public void loadContext(String nomApplication) {
        FileInputStream fichier = new FileInputStream(nomApplication + ".properties");
        props.load(fichier);
        fichier.close();

        String urlDrivers = props.getProperty("jdbc.drivers");
        Class.forName(urlDrivers);
    }
}

class Client {

    public static void main(String[] args) {
        IApplicationContext.INSTANCE.loadContext("%nomApplication%");
    }
}

```

PluginFactory (pour SSO)

Les SSO sont pour l'instant dépendants de la classe matrice qui les implémente du fait que la constante `INSTANCE` est définie de manière concrète dans le code. Pour pouvoir passer facilement d'une implémentation à une autre, nous allons alors introduire le concept de `PluginFactory`.

La classe `PluginFactory` sera une classe et non un SSO (on aurait pu, mais il n'y a pas d'ostracisme outrancier contre les méthodes de classe (et il eu fallu que `PluginFactory` s'invoque elle-même ou sa sœur! Un peu limite !). Elle sera publique car les classes qui en auront besoin seront dans d'autres packages.

Cette classe chargera un fichier `.properties` qui contiendra uniquement des combinaisons Interface – Implémentation.

Lorsque le SSO `INSTANCE` s'initialisera, il demandera à présent à la `PluginFactory` une instance d'implémentation plutôt que d'instancier grâce à un `new` d'une classe « hardcoded ». Celle-ci sera celle spécifiée dans le fichier `.properties`, donc modifiable. C'est ainsi que les classes-Mockup peuvent être remplacées par des classes « réelles » aisément.

La méthode `getPluginFor(...)` retourne un `Object`, car le type de l'instance retournée est inconnu à la compilation ! Il faut caster !

Exemple :

```

public interface SSO {

    public static final SSO INSTANCE = (SSO) PluginFactory.getPluginFor(SSO.class);
}

class SSOImpl1 implements SSO {
}
class SSOImpl2 implements SSO {
}
class SSOImpl3 implements SSO {
}

public class PluginFactory {

    private static Properties props = new Properties();

    static {
        String urlPlugins = IApplicationContext.INSTANCE.getProperty("pluginProps");
        FileInputStream fichier = new FileInputStream(urlPlugins + ".properties");
        props.load(fichier);
        fichier.close();
    }

    public static Object getPluginFor(Class iface) throws PluginFactoryException {
        String implName = props.getProperty(iface.getName());
        try{
            return Class.forName(implName).newInstance();
        } catch (Exception e) { throw new PluginFactoryException(e,...) }
    }
}

```