|  |  |
| --- | --- |
| |  | | --- | |  | |
| C34-VM Programmation Orientée Objet 1  **MODULE 02**  **Introduction à Java et Concepts Orienté Objet – Partie 2**  Automne 2023 |

Table des matières

[1. OO – Principes Fondamentaux de la programmation orientée objet 3](#_Toc142329879)

[1.1 Encapsulation 3](#_Toc142329880)

[Pourquoi? 3](#_Toc142329881)

[1.2 Abstraction 4](#_Toc142329882)

[Pourquoi? 4](#_Toc142329883)

[1.3 Héritage 4](#_Toc142329884)

[Pourquoi? 4](#_Toc142329885)

[1.4 Polymorphisme 5](#_Toc142329886)

[Pourquoi? 5](#_Toc142329887)

[2. OO - Modélisation 6](#_Toc142329888)

[2.1 Les diagrammes de classes 7](#_Toc142329889)

[Les multiplicateurs (liens de relations) 8](#_Toc142329890)

[2.2 Graphique de classe simplifié : modèle conceptuel 9](#_Toc142329891)

[L02A\_Diagrammes de classes UML 9](#_Toc142329892)

[3. JAVA - Les variables 10](#_Toc142329893)

[3.1 Paramètres 10](#_Toc142329894)

[3.2 Valeur de retours d’une méthode 11](#_Toc142329895)

[3.3 Appel de méthodes 11](#_Toc142329896)

[L02B\_Variables et méthodes 11](#_Toc142329897)

[4. JAVA - Les types 12](#_Toc142329898)

[4.1 Catégories de types 12](#_Toc142329899)

[Types primitifs (prédéfinis dans Java) 12](#_Toc142329900)

[Types Non-primitifs 12](#_Toc142329901)

[Le cas particulier de String 12](#_Toc142329902)

[4.2 Les Enveloppeurs (Wrappers) 13](#_Toc142329903)

[Wrappers des types primitifs : 13](#_Toc142329904)

[L02C\_Types 13](#_Toc142329905)

[5. Les modificateurs de non-accès 14](#_Toc142329906)

[5.1 static 14](#_Toc142329907)

[5.2 final 16](#_Toc142329908)

[5.3 Autres modificateurs de non-accès 17](#_Toc142329909)

[L02D\_Modificateurs de non-accès 17](#_Toc142329910)

[TRAVAIL PRATIQUE 2 17](#_Toc142329911)

# OO – Principes Fondamentaux de la programmation orientée objet

La programmation orientée objet est plus que simplement la combinaison des données et actions dans une même structure.

La programmation orientée objet s’appuie sur des principes fondamentaux qui ont pour objectif de faciliter la conception, la maintenance, la sécurité, la rapidité et simplicité de programmation et plus encore.

Ces concepts de base ne sont pas limités à Java, ils sont intégrés à la plupart des langages orienté objet. Ils sont :

* Encapsulation
* Abstraction
* Héritage
* Polymorphisme

## Encapsulation

Consiste à rassembler les données et les actions d’une entité dans une même structure et à contrôler l’accès aux détails de cette structure.

L’accès aux données et les actions qui peuvent y être appliquées ne peuvent être que celles autorisées.

À partir du programme principal, on ne voit pas les détails du code des méthodes d’un objet, ces détails sont « cachés » pour les autres objets.

De plus, à l’aide des **modificateurs d’accès** on peut “protéger” des champs et des méthodes, les isolant du programme principal.

Dans ce contexte, les champs et méthodes sont encapsulées dans la classe.

### Pourquoi?

L’encapsulation permet de cacher les détails du fonctionnement d’un objet aux autres objets.

L’encapsulation permet de rendre des informations confidentielles non disponible à l’extérieur d’une classe.

L’encapsulation permet de contrôler comment les données d’un objet sont traitées (actions permises et interdites).

L’encapsulation permet de contrôler comment les données sont manipulées (validation de champs etc.)

## Abstraction

L’abstraction consiste à cacher les détails d’une partie du code pour en simplifier l’utilisation.

Le code requis pour chaque méthode est conservé dans la classe.

Lorsqu’on utilise les classes on ne voit pas tout ce code à partir du programme principal, le code principal est donc plus net, plus simple.

La complexité du code requis pour réaliser une méthode est cachée dans la classe, permettant au code du programme principal de ne pas en tenir compte.

Tout le code qui se rapporte à un sujet donné se trouve au même endroit : dans la classe.

De plus, on peut regrouper des comportements communs à plusieurs classes dans une classe commune.

### Pourquoi?

Code plus simple et mieux organisé, gestion plus facile

Réduction de la complexité du programme, au moment de la conception et du codage

Maintenance du code plus facile : on peut modifier/corriger des parties du code sans en affecter d’autres

## Héritage

L’héritage permet de configurer des caractéristiques à partir d’un endroit unique et de les utiliser dans des sou-composants.

Avec les sous-classes, une caractéristique commune est configurée dans la classe parente et n’a pas besoin d’être configurée séparément dans chaque sous-classe.

Lorsqu’on a besoin de spécialiser une classe (ajouter des caractéristiques spécifiques à un groupe d’objets), on n’a pas besoin de refaire tout le travail et doubler le code.

### Pourquoi?

Pour simplifier le code en conservant les propriétés d’une classe au moins d’endroits possibles

Profiter de la réutilisation du code

Assurer une cohérence entre des classes de même nature

## Polymorphisme

Consiste à permettre à du code de se comporter différemment dépendant du contexte.

Les classes permettent l’utilisation de méthodes qui portent le même nom, mais qui acceptent des paramètres différents.

On peut par exemple utiliser une méthode appelée **addition** qui se comporterait de façon différente si on tente d’additionner des entiers, des réels ou des chaines de caractères.

### Pourquoi?

Améliorer la réutilisation du code (du point de vue appelant dans ce cas)

Pour simplifier le code

Pour fournir une interface plus uniforme

# OO - Modélisation

L’**analyse** est une des étapes les plus importantes du début d’un projet de programmation. Faite avant de commencer à coder, elle permet de planifier et de distribuer les tâches et d’éviter de refaire des parties du projet.

En programmation orientée objet une partie de l’analyse consiste à identifier les classes, leurs composants et leurs relations.

Autrement dit, qu’est-ce qui sera un objet, quels seront ses membres et comment ces objets interagiront entre eux.

La **modélisation** est un des outils permet de faire cette identification de départ, elle consiste à exprimer la conception d’un projet dans une représentation facile à comprendre, avant de faire du code.

Le **UML** (Unified Modeling Language) un des outils qui permet de faire de la modélisation. UML offre une structure standard de présenter les éléments de conception qui est utilisée et comprise par tous.

UML offre en autres des **diagrammes** pour représenter graphiquement les éléments de conception du logiciel.

En UML, plusieurs types de diagrammes sont utilisés, on se concentre ici sur le **Diagramme de classes**.

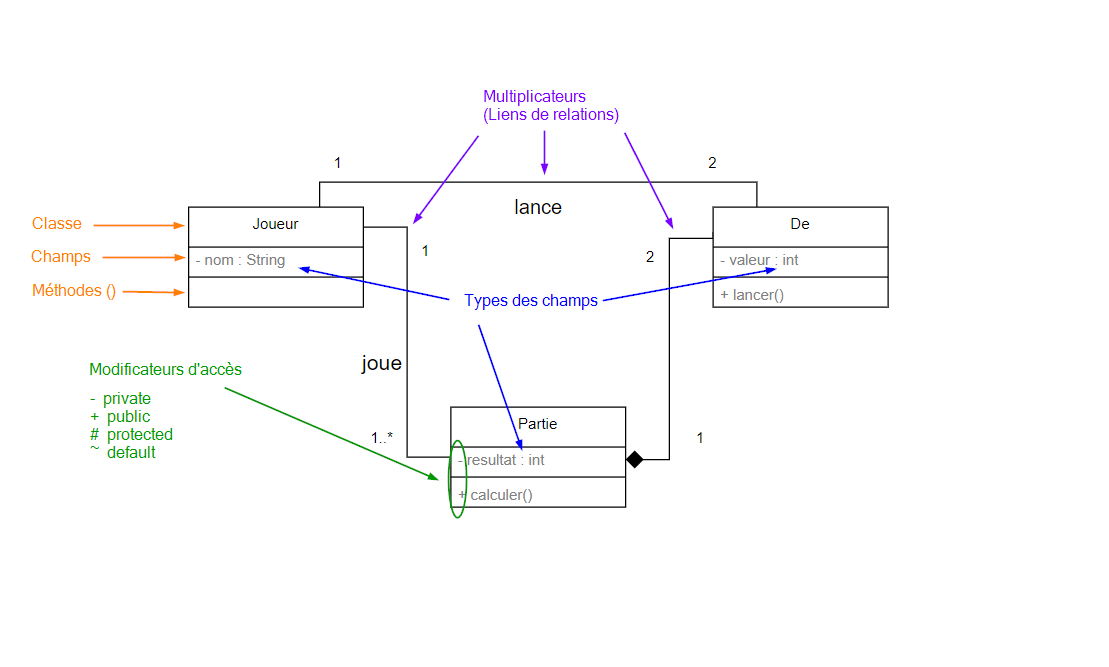
De plus, on créé les diagrammes à partir d’une logique de **modèle conceptuel**, en omettant certains des détails spécifiques pour se concentrer sur les informations et leurs relations.

## Les diagrammes de classes

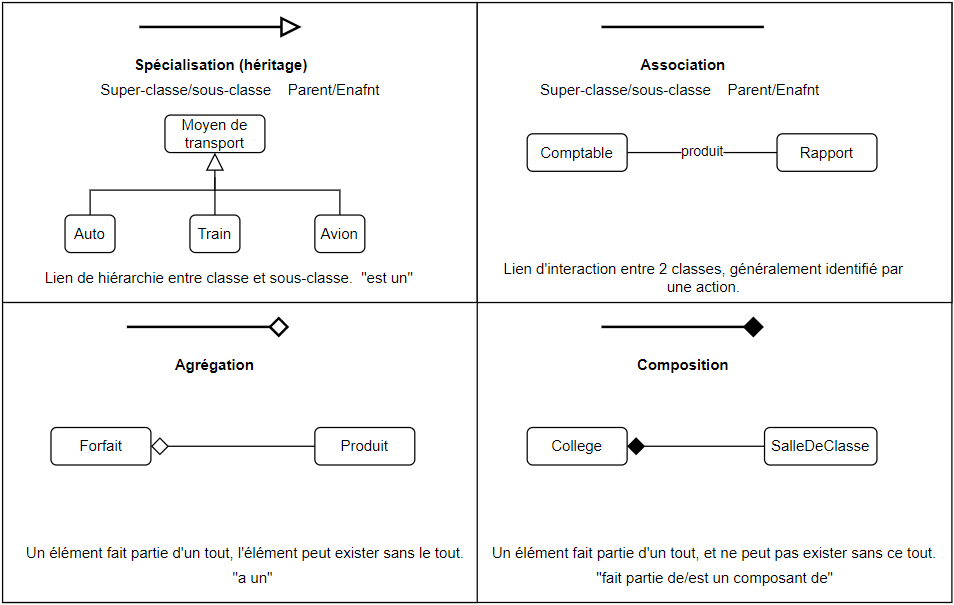
Voici quelques-uns des éléments de base d’un **diagramme de classes UML**.

L’exemple est basé sur la l’énoncé suivant :

Modéliser un jeu de dés où le joueur gagne si la somme de 2 dés est égale à 7 ou 11 (craps). Le joueur peut jouer plusieurs parties.



### Les multiplicateurs (liens de relations)

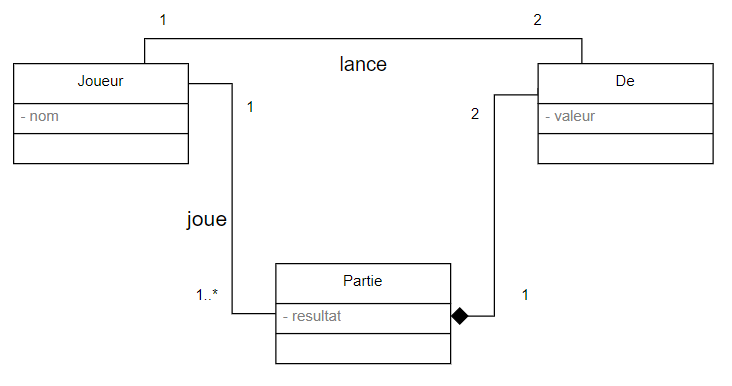


## Graphique de classe simplifié : modèle conceptuel

Pour nos besoins on utilise une version du diagramme selon un **modèle conceptuel**, on laisse de côté les types et les méthodes, pour se concentrer sur :

* Qu’est-ce qui devrait être une classe
* Qu’est-ce qui devrait être un champ d’une classe (et leurs modificateurs d’accès)
* Quels sont les liens entre les classes

Pour notre exemple plus haut, on obtient :



Dans cet exemple, les dés ont une relation de composition avec la partie dans la mesure où les dés n’existent pas sans la partie, ils sont un composant de la partie. Dans une telle application, les dés sont créés et instanciés en même temps qu’une nouvelle partie.

Le joueur et la partie ont une relation d’association, la partie n’est pas un composant du joueur.

Le joueur et les dés ont une relation d’association, ils interagissent ensemble, sans que l’un fasse partie de l’autre.

### L02A\_Diagrammes de classes UML

Le laboratoire L02A vous demande de créer des diagrammes de classes UML à partir d’une mise en situation.

# JAVA - Les variables

Une variable est un espace en mémoire qui peut contenir de l’information.

Déclaration d’une variable :

* *type nomDeVariable;*
* *type nomDeVariable = valeur;*
* *type nomDeVariable1, nomDeVariable2, nomDeVariable3 …… ;*

Une variable doit impérativement être déclarée.

Convention de noms : commence toujours par une minuscule, utilisation de Camel Case

Nom de variable est sensible à la casse

Les **champs** sont une catégorie de variable qui s’appellent :

* **Variable d’instance** (non statique), appartient à une instance de classe, à un objet
* **Variable de classe** (statique, expliqué plus loin), appartient à une classe

, et qui existent tant qu’une instance est initialisée ou à l’appel d’une méthode de classe statique.

Une variable peut être utilisée à l’intérieur du code d’une méthode (comme dans une fonction en programmation fonctionnelle).

Une variable dans une méthode :

* S’appelle une **variable locale**.
* Est supprimée à la sortie de la méthode

## Paramètres

Les **paramètres** passés à une méthode sont aussi une catégorie de variable.

On déclare et on utilise les paramètres sous la forme suivante :

|  |
| --- |
| public void nomDeMethode(typeParametre1 nomParametre1, typeParametre2 nomParametre 2, etc.){  int varAddition = nomParametre1 + nomParametre2;  } |

On appelle une méthode en passant les valeurs des paramètres entre parenthèses :

|  |
| --- |
| int total;  total = unObjet.nomDeMethode(valeurParametre1, valeurParametre2); |

## Valeur de retours d’une méthode

Une méthode peut retourner un résultat ou non.

On doit déclarer le type de la valeur retournée par une méthode, le mot clé **void** indiquant qu’il n’y a pas de valeur retournée.

On déclare le type de la valeur de retour d’une méthode de la façon suivante :

**Exemple 1**

|  |
| --- |
| public void nomDeMethode(){  //Bloc de code  } |

**Exemple 2**

|  |
| --- |
| public int nomDeMethode(int c1, int c2){  int total = c1 + c2;  return total;  } |

Le deuxième exemple montre comment retourner la valeur dans la méthode.

La valeur de retour d’une méthode peut être utilisée dans le code.

**Exemple 1**

|  |
| --- |
| int calcul = unObjet.nomDeMethode(1, 2); |

**Exemple 2**

|  |
| --- |
| System.out.println(unobjet.nomDeMethode(1, 2); |

## Appel de méthodes

Les méthodes peuvent être appelées dans les différents blocs de code, incluant à partir d’autres méthodes, d’une autre classe ou encore de la même classe.

Si une méthode retourne une valeur, elle peut être assignée directement à une variable.

### L02B\_Variables et méthodes

Le laboratoire L02B vous permet de pratiquer les notions de variables et de méthodes.

# JAVA - Les types

Les variables en Java doivent être déclarées avec un type (langage fortement typé).

## Catégories de types

### Types primitifs (prédéfinis dans Java)

* Prédéfinis en Java
* Commencent par une minuscule
* Ne peut pas être null
* N’est pas considéré comme un objet (n’utilise pas **new** pour instancier une variable)
* Les 8 types primitifs :
  + **byte** : entier signé 1 octet
  + **short** : entier signé 2 octets
  + **int** : entier signé 4 octets
  + **long** : entier signé 8 octets
  + **float** : réel 4 octets
  + **double** : réel 8 octets
  + **boolean** : vrai ou faux, 1 bit
  + **char** : caractère, 3 octets, 1 caractère/lettre ou valeur ASCII

Type primitif pas une classe => pas de méthodes (pas de int.to\_string() par exemple)

### Types Non-primitifs

* String, Array, Class, Interface
* Tout autre type défini par le programmeur
* Commence par une majuscule
* Peut être null
* Sont des objets (peuvent être instanciés avec **new**)

### Le cas particulier de String

**String** est un type non-primitif, et une variable de type String est un objet.

Mais il est tout de même prédéfini en Java, avec un support spécial (pas besoin de **new**) et peut presque être considéré et utilisé comme un des types primitifs.

## Les Enveloppeurs (Wrappers)

Classes prédéfinies qui offrent des fonctionnalités supplémentaires aux type primitifs.

**Exemple :**

Classe prédéfinie **Integer** est un wrapper de **int**.

|  |
| --- |
| String nombreTextuel(“42”);  Integer var = new Integer();  var.toString(8);  var.Integer(nombreTextuel); |

### Wrappers des types primitifs :

Character char

Byte byte

Short short

Integer int

Long long

Float float

Double double

Boolean boolean

### L02C\_Types

Le laboratoire L02C vous permet de pratiquer les types en Java.

# Les modificateurs de non-accès

Les **modificateurs de non-accès** ne définissent pas l’accessibilité (visibilité) des variables et des méthodes mais leurs fournissent des propriétés spéciales.

## static

Indique qu’un champ ou méthode existe indépendamment d’une instance (pas besoin de créer un objet).

Appartient à la classe.

Disponible ou exécuté au chargement de la classe.

Peut modifier :

**Classe** – utilisé pour des hiérarchies de classes

**Méthode** – ne peut accéder que des membres statiques d’une classe, au moment du chargement de la classe. Ex : méthode main, toujours chargée sans avoir besoin d’en créer un objet

**Bloc de code** – sert à exécuter du code au moment du chargement de la classe

**Variables**

* une variable membre avec **static** est appelée **variable de classe**
* une variable locale ne peut pas être **static**
* Utilise un seul espace mémoire pour toute la classe au lieu de chaque instance.

Appelé de la façon suivante :

* Champ : NomDeLaClasse.NomDuChamp
* Méthode : NomDeLaClasse.NomDeMethode()

**Exemple**

Soit une classe **Client** pour manipuler des informations de clients.

Chaque client obtient un numéro d’identification unique (**idClient**) qui est incrémenté à chaque nouveau client.

L’identifiant le plus récemment utilisé (**idClientCourant**) est un bon candidat pour un champ statique.

L’information est conservée 1 fois pour la classe indépendamment des instances.

|  |
| --- |
| **CLASSE Client**  **package exemplestatic1.module02;  public class Client {  public static int *idClientCourant*; *//champ - variable de classe - dernier idClient utilisé* private int idClient; *//champ - variable d'instance* private String nomClient; *//champ - variable d'instance   //Méthode qui initialise un objet Client* public void setClient(String parNomClient) {  *idClientCourant*++; *//Définir le prochain idClient  //Pas conservé dans l'objet mais par la classe* this.idClient = *idClientCourant*; *//Assigner le prochain idClient à cet objet* this.nomClient = parNomClient; *//Assigner le nom de client passé en paramètres* }   public String getNomClient() {  return this.nomClient;  }   public int getIdClient() {  return this.idClient;  }   public int getIdClientCourant() {  return this.*idClientCourant*;  } }**  **CLASSE PRINCIPALE - GestionCLient**  **package exemplestatic1.module02;  public class GestionClient {  public static void main(String[] args){  *//Initialiser la valeur de la variable de classe - indépendante des objets* Client.*idClientCourant* = 1000;   *//La variable de classe n'a pas besoin d'un objet Client pour être utilisée* System.*out*.println("L’ID de client courant est " + Client.*idClientCourant* + "\n");     *//Créer 3 objets Client* Client client1 = new Client();  Client client2 = new Client();  Client client3 = new Client();   *//Initialiser les 3 objets clients* client1.setClient("Harry Granger");  client2.setClient("Hermione Weasley");  client3.setClient("Ron Potter");   *//Chaque objet client a reçu un idClient, basé sur la valeur de la variable de classe  //idClientCourant (static), incrémentée de 1 à chaque nouveau client* System.*out*.println("L’ID de " + client1.getNomClient() + " est " + client1.getIdClient());  System.*out*.println("L’ID de " + client2.getNomClient() + " est " + client2.getIdClient());  System.*out*.println("L’ID de " + client3.getNomClient() + " est " + client3.getIdClient() + "\n");   *//La variable de classe est à jour* System.*out*.println("L’ID de client courant est " + Client.*idClientCourant* + "\n");    *//L'idClientCourant est le même pour tous les objets, parcequ'il n'est pas relié aux objets mais à la classe  //Lorsqu'on a manipulé les objets client1 et client2, l'idClientCourant était 1001 et 1002* System.*out*.println("L’idClientCourant pour " + client1.getNomClient() + " est " + client1.getIdClientCourant());  System.*out*.println("L’idClientCourant pour " + client2.getNomClient() + " est " + client2.getIdClientCourant());  System.*out*.println("L’idClientCourant pour " + client3.getNomClient() + " est " + client3.getIdClientCourant());   } }**  **RÉSULTAT**  L’ID de client courant est 1000  L’ID de Harry Granger est 1001  L’ID de Hermione Weasley est 1002  L’ID de Ron Potter est 1003  L’ID de client courant est 1003  L’idClientCourant pour Harry Granger est 1003  L’idClientCourant pour Hermione Weasley est 1003  L’idClientCourant pour Ron Potter est 1003 |

## final

Interdit la modification de la valeur d’une variable, méthode ou classe.

Peut modifier :

**Classe** – ne peut pas être étendu, héritage bloqué (héritage expliqué plus loin)

**Méthode** – ne peut pas être surchargée dans une sous-classe (sous-classes expliquées plus loin)

**Variables** (de classe, d’instance, argument de méthode)

* la valeur de la variable ne peut être modifiée
* la variable ne peut être initialisée qu’une fois
* représente une CONSTANTE
* convention de nomenclature en majuscules ave traits de soulignement entre les mots. Ex : NOMBRE\_PI.

Une variable avec **static** et **final** est la façon typique de définir une **constante** pour une classe.

Lorsqu’une valeur est fixe pour un objet, elle devrait être définie comme une constante (champ static et final).

La **convention de nom** pour les **constantes** :

* Tout en majuscule
* Mots séparés par des barres de soulignement (underscore).
* Ex : TAXE\_FEDERALE, TAXE\_PROVINCIALE, RABAIS\_PROMO, PRIX\_FORFAIT1 etc.

## Autres modificateurs de non-accès

**abstract** – pour une classe indique une classe qui ne peut être instanciée, pour une méthode, indique une méthode sans définition, qui sera plutôt définie dans une sous-classe.

**synchronized** – interdit l’accès à une méthode par plusieurs threads en même temps.

**transient** – indique qu’un membre d’une classe ne doit pas être stocké en mémoire ou envoyé sur le réseau avec le reste de l’objet. Utilisé par exemple pour des mots de passe.

**scrictfp** – assure la production d’un résultat d’opération point flottant identique sur toute plateforme

**native** – permet d’écrire du code dans une méthode utilisant un autre langage que Java (Ex : en C)

### L02D\_Modificateurs de non-accès

Le laboratoire L02D vous permet de pratiquer les modificateurs de non-accès.

### TRAVAIL PRATIQUE 2

Le TP2 vous demande de mettre en pratique les éléments appris jusqu’à maintenant.

NOTES :

La dernière partie du TP2 vous invite à préparer des tests unitaires. C’est une façon de vérifier le comportement du code en lui injectant une série de valeurs et en vérifiant es résultats.

Le TP utilise JUnit, un cadre open-source conçu pour ce genre de tests.

Suivez les instructions fournies dans le TP pour mettre en place des tests JUnit pour votre code.