Consolidations des concepts

Création de classes

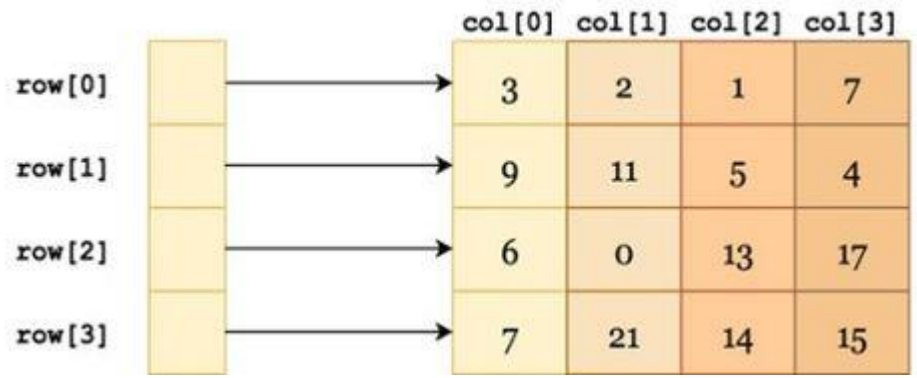
1. Créer une classe abstraite Personne :
   Nom (String),
   Prenom (String),
   Age (int),
   Constructeur pour l'initialisation
   Méthode abstraite afficherRôle()
2. Créer des sous-classes Etudiant et professeur qui hérite de la classe Personne, et qui utilise le mécanisme du polymorphisme pour afficher le rôle de chaque personne enregistrée (pour cela, une méthode qui m'affiche le nom de chaque personne et son rôle au sein de l'école).
3. Ajouter l'attribut « Appreciation » de type String dans la classe Etudiant. Et un attribut « MatiereEnseignee » de type String dans la classe Professeur.
4. Créer une enum qui enregistre les spécialités enseignées à l'école (IIR, GF, GI, GC/BTP, GESI, IAII). Ajouter un attribut à la classe Etudiant de type enum pour associer une spécialité à un étudiant.

```
int[] tableauEntier = {1, 2, 3, 4, 5};
String[] tableauChaine = {"Bonjour", "le", "monde"};

int[] tableauEntier = new int[] {1, 2, 3, 4};
String[] tableauChaine = new String[] {"Bonjour", "le", "monde"};

int[] tableauEntier = new int[5];
String[] tableauChaine = new String[3];
```

la taille d'un tableau est donné à sa création et ne peut plus être modifiée. Il n'est donc pas possible d'ajouter ou d'enlever des éléments à un tableau. Dans ce cas, il faut créer un nouveau tableau avec la taille voulue et copier le contenu du tableau d'origine vers le nouveau tableau.

les éléments d'un tableau sont initialisés avec une valeur par défaut

Les tableaux multi-dimensionnels :



3	2	1	7
9	11	5	4
6	0	13	17
7	21	14	15

Méthode	Description
`static int binarySearch(type[] a, type key)`	Rechercher dans le tableau spécifié la valeur de key spécifiée à l'aide de l'algorithme de recherche binaire.
`static boolean equals(type[] a, type[] a2)`	Renvoyer true si les deux tableaux du même type spécifiés sont égaux.
`static void fill(type[] a, type val)`	Affecter la valeur spécifiée à chaque élément du tableau spécifié
`static void sort(type[] a)`	Trier le tableau spécifié en ordre croissant
`static void sort(type[] a, int fromIndex, int toIndex)`	Trier la plage spécifiée du tableau en ordre croissant
`static void parallelSort(type[] a)`	Trier le tableau spécifié en ordre croissant à l'aide d'un tri en parallèle.
`static void parallelSort(type[] a, int fromIndex, int toIndex)`	Trier la plage spécifiée du tableau en ordre croissant en utilisant le tri en parallèle
`static List asList(type ... a)`	Renvoyer une liste de taille fixe contient les éléments des tableaux spécifiés.
`static type[] copyOf(type[] originalArray, int newLength)`	Copier le tableau spécifié, en tronquant ou en remplissant avec la valeur par défaut (si nécessaire) afin que la copie ait la longueur spécifiée.
`static type[] copyOfRange(type[] originalArray, int fromIndex, int endIndex)`	Copier la plage spécifiée du tableau spécifié dans un nouveau tableau.
`static String toString(type[] originalArray)`	renvoyer une représentation sous forme de chaîne du contenu du tableau spécifié.

Comparaison entre equals() et compareTo()

Méthode	Objectif	Retourne	Utilisation
equals(Object o)	Vérifie l'égalité de contenu	boolean	Tester si deux objets sont égaux
compareTo(T o)	Compare deux objets pour l’ordre	int (-1, 0, 1)	Utilisé dans le tri et l’ordre

Tableaux

1. Créer une méthode dans la sous-classe étudiant qui permet d'enregistrer les notes de l'étudiants
2. Créer un tableau de double, de dimension définie par l'utilisateur, qui servira pour enregistrer les notes des étudiants. Remplissez le tableau à l'aide de l'utilisateur.
3. Afficher la moyenne des notes ➔ la somme / le nombre de notes
4. Afficher le tableau sous format d'une chaine de caractères
5. Créer un autre tableau de la même dimension que le premier tableau, il servira pour enregistrer les notes du deuxième semestre. Remplissez le tableau à l'aide de l'utilisateur.
6. Comparer les deux tableaux. Si les tableaux sont égaux, afficher une erreur ➔ l'utilisateur s'est trompé de notes (Il nous a donné les notes du premier semestre et non pas du deuxième semestre).
7. Faites le tri des tableaux.
8. Cherchez moi où se trouve la note X dans le tableau, X donné par l'utilisateur.
9. Transformer les tableaux en liste.
10. Créer un tableau multi-dimensionnel de taille 2*3*2, demandez à l'utilisateur de remplir le tableau puis affichez-le.

Chaines de caractères

1. Créer une méthode testErreurNom() dans la classe Etudiant qui contrôle l'égalité entre nom et prenom. Si les deux valeurs sont égaux, on affiche erreur et on demande à l'utilisateur de retaper le nom et le prénom.
2. Calculer la longueur du nom et prénom de l'étudiant.
3. Modifier les attributs nom et prénom en un seul attribut qu'on va nommer nomComplet. Séparer entre nom et prénom dans le nomComplet.
4. Créer une méthode qui permet de manipuler l'appréciation faites par les enseignants sur l'étudiant. Chercher dans le texte s'il y a un « bien » ou un « mauvais ». Retourner « positif » s'il y a le mot « bien » dans le texte, et « négatif » s'il y a le mot « mauvais » dans le texte.
5. Afficher tout le texte écrit après le mot « bien » ou le mot « mauvais » donné dans l'appréciation de l'enseignant.

Est-ce qu’un mot existe dans une chaine de caractère ?

Méthode	Description
contains()	Vérifie si une chaîne contient une sous-chaîne.
indexOf()	Retourne l'index de la première occurrence.

- La classe String, comme certaines classes en java, ne nécessite par d’instanciation pour l’utilisation (notamment avec l’autoboxing, les enums, les méthodes de fabrique et les classes anonymes). On peut créer et utiliser des objets sans instanciation explicite avec new.
- Ne pas instancié avec new ne signifie pas que l’on ne peut pas instancier un String : il y a simplement deux manières différentes de créer des objets String en Java, et chacune a ses propres implications.
    1. Quand on crée une chaîne de caractères comme ceci : String str1 = "Hello"; Java place "Hello" dans un espace mémoire spécial appelé *pool de chaînes* (ou *string pool*). Dans ce cas, si une autre chaîne de caractères avec la même valeur ("Hello") existe déjà dans le pool, Java réutilisera cette instance au lieu d'en créer une nouvelle. C'est un moyen d'optimiser la mémoire, car il évite de créer des objets String identiques en double. String str1 = "Hello"; String str2 = "Hello"; Ici, str1 et str2 référencent le même objet "Hello" dans le pool de chaînes.
    2. On peut aussi créer un objet String en utilisant new : String str3 = new String("Hello"); Dans ce cas, un nouvel objet String est créé dans la mémoire, même si un objet identique existe dans le pool. Cela signifie que même si str3 contient la même valeur que str1 ou str2, il fait référence à un objet distinct.

```
String str1 = "Hello"; String str3 = new String("Hello");
System.out.println(str1 == str3); // Affiche false, car ce sont deux objets différents
```

Pourquoi utiliser les littéraux String plutôt que new String ?

- **Optimisation de la mémoire** : En utilisant des littéraux, on profite du pool de chaînes, ce qui permet de réutiliser les chaînes et de réduire la consommation mémoire.
- **Performance** : Instancier une chaîne avec new est plus coûteux que l'utilisation d'un littéral, car chaque appel à new crée un nouvel objet String en mémoire.

En résumé, on utilise généralement des littéraux String parce qu'ils sont plus efficaces en termes de mémoire et de performance, et Java les optimise en les plaçant dans le pool de chaînes.

L'autoboxing:

L'autoboxing en Java est le processus automatique par lequel le compilateur convertit une valeur primitive en son équivalent *wrapper class* (classe enveloppe). Ce processus inverse s'appelle **unboxing**.

Java a été conçu avec deux types principaux : les types primitifs (comme int, double, etc.) pour des raisons de performance, et les classes wrapper (Integer, Double, etc.) qui permettent de manipuler ces types primitifs comme des objets.

L'autoboxing et l'unboxing facilitent l'utilisation des types primitifs dans des contextes où des objets sont requis, comme dans les collections (List, Map, etc.), qui ne peuvent pas stocker de types primitifs directement.

L'autoboxing et l'unboxing simplifient donc le code et l'interaction entre les types primitifs et les objets, mais ils doivent être utilisés avec soin pour éviter les problèmes de performance et les erreurs de NullPointerException (avec l'unboxing, si un wrapper null est déballé en primitif, cela déclenchera une NullPointerException).

Stringbuilder

StringBuilder est une classe en Java qui permet de manipuler des chaînes de caractères de manière efficace en environnement *non synchronisé* (c'est-à-dire sans verrouillage pour les threads). Contrairement à String, qui est immuable (non modifiable une fois créée), StringBuilder est mutable, ce qui signifie que son contenu peut être modifié sans créer de nouveaux objets. Cela en fait un excellent choix pour les opérations qui nécessitent de nombreuses modifications, telles que les concaténations répétées ou les manipulations de chaînes dans des boucles.

Mutable :

•Avec StringBuilder, les opérations comme l'ajout, la suppression, ou la modification de caractères se font directement sur l'objet. Cela évite la création de nouvelles instances, contrairement aux chaînes String qui nécessitent une nouvelle instance chaque fois qu'elles sont modifiées.

•Cette mutabilité améliore les performances pour les opérations répétées.

Non synchronisé :

•StringBuilder n'est pas synchronisé, ce qui signifie qu'il est plus rapide que StringBuffer en environnement *non concurrent*. En revanche, cela le rend non sûr dans un contexte multithread.

•Cela le rend particulièrement performant pour des opérations dans des environnements où un seul thread est utilisé, ou lorsque l'accès concurrent n'est pas un problème.

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World"); // Résultat : "Hello World«

sb.insert(5, ", Java"); // Résultat : "Hello, Java World"

sb.delete(5, 10); // Résultat : "Hello World"

sb.replace(6, 11, "Java"); // Résultat : "Hello Java"

sb.reverse(); // Résultat : "avaJ olleH"

String result = sb.toString(); // convertit le StringBuilder en une chaîne de caractères (String).

StringBuilder sb = new StringBuilder();
for (int i = 0; i < 5; i++) {
   sb.append("Numéro ").append(i).append(", ");
}
System.out.println(sb.toString()); // Sortie : "Numéro 0, Numéro 1, Numéro 2, Numéro 3, Numéro 4, "
```

StringBuffer

Les objets de type String ne sont pas modifiables, mais il est possible de les employer pour effectuer la plupart des manipulations de chaînes.
Java dispose d'une classe **StringBuffer** destinée elle aussi à la manipulation de chaînes, mais dans laquelle les objets sont modifiables.
On peut créer un objet de type StringBuffer à partir d'un objet de type String. Il existe des méthodes :

- De modification d'un caractère de rang donné :
- setCharAt,
- D'accès à un caractère de rang donné : charAt,
- D'ajout d'une chaîne en fin : la méthode append accepte des arguments de tout type primitif et de type String,
- D'insertion d'une chaîne en un emplacement donné : insert,
- De remplacement d'une partie par une chaîne donnée : replace
- De conversion de StringBuffer en String : toString.

Voici un programme utilisant ces différentes possibilités :

```
class TestclassstrinBuffer
{ public static void main (String args[])
{ String ch = "la java" ;
StringBuffer chBuf = new StringBuffer (ch) ;
System.out.println (chBuf) ;
chBuf.setCharAt (3, 'J'); System.out.println (chBuf) ;
chBuf.setCharAt (1, 'e') ; System.out.println (chBuf) ;
chBuf.append (" 2") ; System.out.println (chBuf) ;
chBuf.insert (3, "langage ") ; System.out.println (chBuf) ;
}
}
```

```
la java
la Java
le Java
le Java 2
le langage Java 2
```

Manipulation de dates

- Utilisation de l'API java.time, introduite avec Java 8, qui est maintenant la manière recommandée de gérer les dates et heures :

```
import java.time.LocalDate; import java.time.Period;
```

1. Ajouter dans votre classe étudiant l'attribut date de naissance (sous format JJ-MM-AAAA), la date d'obtention du bac (sous format JJ-MM-AAAA) et la date d'entrée à l'EMSI (sous format JJ-MM-AAAA).
2. Déduisez l'âge de l'étudiant.
3. Comparer la date d'obtention du bac et la date d'entrée à l'école, déduisez si l'étudiant à commencer son parcours universitaire avec l'EMSI ou pas.
4. Ajouter un mois et trois semaines à la date d'aujourd'hui et annoncez la date du contrôle.

```
// Ajouter un mois
LocalDate datePlusUnMois = dateActuelle.plusMonths(1);

// Ajouter deux semaines supplémentaires
LocalDate dateFinale = datePlusUnMois.plusWeeks(3);
```

Manipulation de dates

5. Donner le nombre de jours entre la date du contrôle et la date d'aujourd'hui, annoncez combien de jours restent pour le contrôle.

6. Afficher la date du contrôle sous différents formats : format long (jour mois année), format court (jour/mois/année) et format ISO (année-mois-jour).

```
DateTimeFormatter format1 = DateTimeFormatter.ofPattern("dd/MM/yyyy");
DateTimeFormatter format2 = DateTimeFormatter.ofPattern("MMMM dd, yyyy");
DateTimeFormatter format3 = DateTimeFormatter.ofPattern("E, MMM dd yyyy");
```

```
// Afficher la date dans différents formats
System.out.println("Format (dd/MM/yyyy) : " + dateActuelle.format(format1));
System.out.println("Format (MMMM dd, yyyy) : " + dateActuelle.format(format2));
System.out.println("Format (E, MMM dd yyyy) : " + dateActuelle.format(format3));
```

Les enums

```
public enum Semaine_jours {
        LUNDI(8),
        MARDI(8),
        MERCREDI(5),
        JEUDI(8),
        VENDREDI(8),
        SAMEDI(0),
        DIMANCHE(0);

        private int nbHeures;

        public Semaine_jours(int nb) { nbHeures = nb; }
        public int getNbHeures() { return nbHeures; }
        public String action() {
                switch(this) {
                        case SAMEDI : return «repos";
                        case DIMANCHE : return «Jardin";
                        default : return « Bosser";
                }
        }
}
```

Les enums

```
public class Principal {
	public static void main(String[] arg) {
		Semaine_jours unJour = Semaine_jours.SAMEDI;
		System.out.println("numero de " + unJour + " : " + unJour.ordinal());
		// la méthode ordinal est une méthode définie dans la classe java.lang.Enum et qui renvoie le numéro
			d'ordre dans l'énumération, en commençant à la  valeur 0.
		System.out.println(unJour + " : " + unJour.action());
		for (Semaine_jours jour : Semaine_jours.values())
			System.out.println("nombre d'heures de " + jour + " : " + jour.getNbHeures());
	}
}
```

```
numero de SAMEDI : 5
SAMEDI : repos
nombre d'heures de LUNDI : 8
nombre d'heures de MARDI : 8
nombre d'heures de MERCREDI : 5
nombre d'heures de JEUDI : 8
nombre d'heures de VENDREDI : 8
nombre d'heures de SAMEDI : 0
nombre d'heures de DIMANCHE : 0
```

Les classes immutables

- Une classe **immutable** (ou "immuable" en français) est une classe dont les objets, une fois créés, ne peuvent plus être modifiés. Cela signifie que toutes les propriétés de l'objet sont définies au moment de sa création et ne peuvent pas être changées ensuite. En Java, une fois qu'un objet immuable est créé, ses données internes ne peuvent être modifiées d'aucune manière, ce qui garantit une grande sécurité et une meilleure gestion de la concurrence.
- La classe String en Java est un exemple bien connu de classe immuable. Une fois qu'un objet String est créé, vous ne pouvez plus changer sa valeur. D'autres classes immuables incluent Integer, Double, LocalDate et LocalDateTime.
- Les classes immuables présentent de nombreux avantages :
    1. **Sécurité** : Puisque les objets immuables ne changent jamais après leur création, ils sont sûrs dans un environnement concurrentiel (multi-threading). Les threads peuvent partager des objets immuables sans risques de modifications inattendues.
    2. **Facilité de maintenance** : Les classes immuables sont plus faciles à tester et à utiliser car elles réduisent la complexité liée aux états variables des objets.
    3. **Meilleure optimisation mémoire** : Les objets immuables peuvent être réutilisés ou stockés en cache pour réduire la consommation de mémoire.

Les classes immutables

Les principales règles pour créer une classe immuable :
1.Déclarez la classe comme final pour éviter qu'elle ne soit étendue, car une sous-classe pourrait introduire des comportements modifiables.
2.Déclarez tous les champs comme final pour garantir qu'ils ne sont affectés qu'une seule fois (au moment de la création de l'objet).
3.Ne fournissez pas de méthodes setter pour empêcher la modification des champs après la création de l'objet.
4.Initialisez tous les champs via le constructeur. Utilisez un constructeur pour définir les valeurs initiales des champs, et ne permettez aucun changement ultérieur.
5.Si un champ est une référence à un objet mutable (comme un tableau ou une liste), créez une copie défensive de cet objet pour empêcher des modifications indirectes.

```
public final class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

L’encapsulation

En Java, l’encapsulation est réalisée par :
1.Déclaration des attributs comme private : Cela restreint l'accès aux données aux seules méthodes de la classe.
2.Utilisation de méthodes public pour accéder et modifier les données : Les méthodes getter et setter permettent un accès contrôlé.

Avantages de l’encapsulation
1.Protection des données : En restreignant l'accès direct aux données, on protège l'objet contre des modifications imprévues ou non sécurisées.
2.Contrôle des accès : On peut ajouter des validations ou des conditions pour contrôler la manière dont les données sont modifiées (comme dans l'exemple de deposer() et retirer()).
3.Facilité de maintenance : Les classes encapsulées sont plus simples à modifier et à maintenir, car leurs détails d'implémentation sont isolés.
4.Flexibilité : On peut modifier l’implémentation interne de la classe sans affecter le reste du programme tant que l'interface publique (les méthodes accessibles) reste la même.

```
public class CompteBancaire {
    // Attribut privé
    private double solde;
    // Constructeur
    public CompteBancaire(double
soldeInitial) {this.solde = soldeInitial;}
    // Getter pour obtenir le solde
    public double getSolde() {return solde;}
    // Setter pour ajouter de l'argent
    public void deposer(double montant) {
        if (montant > 0) {solde += montant;}
        else {System.out.println("Montant
                invalide");}
    }
    // Setter pour retirer de l'argent
    public void retirer(double montant) {
        if (montant > 0 && montant<= solde){
            solde -= montant;}
        else { System.out.println("Montant
                invalide ou insuffisant");
        }
    }
}
```