**1) Les Types et les Définition en Java :**

Les types dans java sont deux type primitifs et wrapper class des classes enveloppes (Integer, Double ) , l’utilité change en fonction de l’utilisation les type primitifs dont généralement utilisé dans des condition des stockage du data vu que il prennent pas assez d’espace comme un objet de type wrapper par contre l’utilité des wrapper class consiste a les utilisé dans des generics contexte comme les arraylist, hashmap ….

l’autoboxing et l’unboxing permet de transformer automatiquement une variable de type primitif en un objet de type wrapper et l unboxing c’est l’inverse

String montexte = "10"; Integer monnombre = new Integer(montexte); int i = monnombre.intValue();

**2) Le Safe Casting**:

pour caster une variable de type predifnit vers un autre type peut engendré des exceptions tel que ClassCastException pour eviter cette exception on peut utiliser instanceof

if (animal instanceof Cat) { Cat cat = (Cat) animal; cat.meow();}

else if (animal instanceof Dog) { Dog dog = (Dog) animal; dog.woof(); }

**3) La Gestion Avancée des Classes et Objets**

structure des classes efficace : Access Modifier

**Abstract**: il y’a au moins une methode abstraites et ne peut pas etre instancié

**Final**: La class ne peut pas etre modifier , pas de child class

**Private**: la class n’est accessible qu’a partir le fichier

**Public**: La classe est accessible partout

**Access modifier for methods :**

**public :** la methode est accessible aux autres method de autres classes

**private**: l usage est reserve aux autre methodes de la meme classe

**protected**: la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous-classes

**fina l**: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).

**synchronized**:

**native**:

**la surcharge des methods**:permet de définir plusieurs fois une même méthode avec des arguments différents

**4) Les Génériques :** sont un ensemble de caractéristique du langage liées a la définition

5) **Api Stream :**

Un stream(flux) est une représentation d’une séquence sur laquelle il est possible d’appliquer des opérations. Cette API a deux principales intérêts

1 : Elle permet d’effectuer les opérations sur une séquence sans utiliser de structure de boucle.

2 : Les opérations sur les streams sont réalisées en flux (d’où leur nom) ce qui limite l’empreinte mémoire nécessaire

La collecte permet de créer un nouvelle collection à partir d’un stream. Pour cela, il faut fournir une implémentation de l’interface Collector. 

**List<String> liste = Arrays.asList("une chaine", "une autre chaine", "encore une chaine");**

**List<String> autreListe = liste.stream().collect(Collectors.toList());**

## Le filtrage

Une opération courante sur un stream consiste à appliquer un filtre pour éliminer une partie de ses éléments. Pour, cela on peut utiliser la méthode Stream.filter.

**List<Voiture> liste = Arrays.asList(new Voiture("citroen"), new Voiture("audi"), new Voiture("citroen"));**

**List<Voiture> sansCitroen = liste.stream() .filter(v -> !v.getMarque().equals("citroen")) .collect(Collectors.toList());**

**System.out.println(sansCitroen.size()); // 1**

**5) Design Pattern :**

**Factory :**La fabrique permet de créer un objet dont le type dépend du contexte  cet objet fait partie d'un ensemble de sous-classes. L'objet retourné par la fabrique est donc toujours du type de la classe mère mais grâce au polymorphisme les traitements exécutés sont ceux de l'instance créée. Une Factory est une classe qui instancie des objets d'une autre classe

**6) ExecutorService**

L’interface ExecutorServicedéfinit une façon de soumettre des tâches, puis de récupérer leurs résultats sour forme de Future<?>. Elle a aussi des méthodes pour arrêter le pool de threads

On peut instancier directement le ThreadPoolExecutor, ou passer par les méthodes de fabrique d'Executors. On passe des paramètres de dimensionnement et de file d’attente au constructeur.

var executor = new ThreadPoolExecutor(

10, 20,

10, TimeUnit.SECONDS,

new LinkedBlockingQueue<Runnable>());

Dans l’ordre, les paramètres sont les suivants :

* corePoolSize = taille du pool principal
* maximumPoolSize = taille maximale du pool
* keepAliveTime et unit = durée d’inactivité d’un thread avant son éviction
* workQueue = file d’attente des tâches

## Classes et interfaces

Les couples classe / interface classiques sont les suivants :

Collection<String> col = new ArrayList<>();

List<String> list = new ArrayList<>();

Set<String> set = new HashSet<>();

Map<String, String> map = new HashMap<>();

L’utilisation de classes anciennes, comme Vector ou Hashtable, est généralement à proscrire.

Avec le JDK 11, la façon la plus simple d’instancier une collection est d’utiliser les nouvelles méthodes de fabrique. Avec ces méthodes, on ne se préoccupe plus du type concret de la collection obtenu, il faut juste savoir qu’elle n’est pas modifiable.

List<String> list = List.of("AZERTY", "QSDFGH", "WXCVBN");

Et ça fonctionne aussi pour créer une Map.

Map<String, String> map = Map.of(

"A", "AZERTY",

"Q", "QSDFGH",

"W", "WXCVBN"

);

## Boucles for each

Depuis le JDK 5, une nouvelle forme de boucle for, surnommée « for each » a été introduite. Elle est beaucoup plus pratique que l’ancienne forme, mais présente des contraintes lorsqu’on veut manipuler la collection qu’on parcourt car l’itérateur n’est pas accessible.

for (Type var : col) {

col.remove(var); // Ceci est interdit !!!

}

Le problème rencontré est une ConcurrentAccessException lorsqu’on supprime un élément de la collection en cours d’itération. L’ancienne boucle, du fait qu’elle sépare distinctement la collection de son itérateur, permet d’ajouter ou de retirer depuis le corps de la boucle de éléments de la collection. Ceci fonctionne à condition de la faire via l’itérateur et non directement sur la collection.

for (Iterator<Type> it = col.iterator();it.hasNext();) {

Type var = it.next();

...

col.remove(var); // Ceci est toujours interdit

it.remove(); // Ceci est autorisé

...

}

## Tri

Le tri d’une liste se fait en passant la liste à trier à la méthode java.util.Collections.sort(list). Cette méthode trie les objets de la liste selon le résultat de leurs méthodes compareTo(). Cela signifie donc que ces objets doivent implémenter l’interface Comparable.

Collections.sort(list);

Pour trier des listes d’objets qui n’implémente pas cette interface, on peut choisir d’utiliser un Comparator et en passer une instance à la méthode java.util.Collections.sort(list, comparator).

## Stream et lambda

Avec le JDK 8, l’introduction de l’API Stream a changé par mal de choses dans la manipiulation des collections.

newCol = col.stream()

.filter(...)

.sort()

.toList();

Pour les comparateurs, la notation lambda simplifie les choses, puisque Comparator est une interface fonctionnelle.

Comparator comp = (Person p1, Person p2) -> p2.getAge() - p1.getAge();

Et le JDK 8 a introduit des méthodes qui facilite l’écriture de comparateurs.

Comparator comp = Comparator.comparing(Person::getAge).reversed();

## Implémentations de List

Comparée à LinkedList, ArrayList presque toujours meilleure, quelle que soit la taille de la liste. Le seul cas où LinkedList est éventuellement meilleure, c’est quand on modifie les éléments intermédiaires de la liste : ajout ou suppression d’objets en début ou au milieu de la liste.

java.util.concurrent.CopyOnWriteArrayList est une variante thread-safe de java.util.ArrayList. Chaque méthode de modification fait une copie du tableau interne.

## Implémentations de Set

L’implémentation par défaut est HashSet.

Avec LinkedHashSet, les objets sont itérés dans l’ordre de leur ajout.

TreeSet implémente SortedSet qui, comme sont nom l’indique, stocke les objets triés dans leur ordre naturel ou selon un comparateur.

EnumSet est utilisé pour les types énumérés. Il est meilleur pour ça parce qu’il ne passe pas par equals() mais ==.

Il y a deux implémentations thread-safe :

* CopyOnWriteArraySet fait une copie de son tableau interne à chaque modification,
* ConcurrentSkipListSet implémente SortedSet.

Il y a une troisième option, à partir d’un ConcurrentHashMap.

Set<String> set = ConcurrentHashMap.newKeySet();

## Implémentations de Map

L’implémentation par défaut est HashMap.

Avec LinkedHashMap, les objets sont itérés dans l’ordre de leur ajout.

TreeMap implémente SortedMap qui, comme sont nom l’indique, stocke les clés triées dans leur ordre naturel ou selon un comparateur.

EnumMap est utilisé pour les clés énumérés. IdentityHashMap compare aussi les clés par ==, ce qui est performant mais contraire au contrat de Map.

Les implémentations thread-safe :

* ConcurrentHashMap est l’implémentation par défaut,
* ConcurrentSkipListMap implémente SortedMap.