# MethodFormation JAVA 8

Table des matières

[MethodFormation JAVA 8 1](#_Toc47510759)

[Méthode de refactoring 6](#_Toc47510760)

[1- Renommer les variables 6](#_Toc47510761)

[2- Dissocier les boucles for utilisées dans la même méthode 6](#_Toc47510762)

[Avant 6](#_Toc47510763)

[Après : 6](#_Toc47510764)

[3- Méthode d’extraction des logiques communes présentes dans différentes méthodes 8](#_Toc47510765)

[Avant 8](#_Toc47510766)

[Apres 8](#_Toc47510767)

[4- Transformation des boucles en utilisant la librairie Stream API 9](#_Toc47510768)

[Avant 10](#_Toc47510769)

[Apres 10](#_Toc47510770)

[5- Déplacement de méthode dans une classe dédiée 10](#_Toc47510771)

[6- Renommage des variables dû aux méthodes de partours des objets, déplacés dans la classe Season 10](#_Toc47510772)

[7- Renommage des méthodes booléènes de vérification 11](#_Toc47510773)

[Avant 11](#_Toc47510774)

[Apres 11](#_Toc47510775)

[Avant 11](#_Toc47510776)

[Après 12](#_Toc47510777)

[Compléments de refactorisation 13](#_Toc47510778)

[Utiliser le pattern Factory pour initialiser le constructeur 13](#_Toc47510779)

[Remplacer le conditionnel par du polymorphisme 14](#_Toc47510780)

[Supprimer les assignements de paramètres 16](#_Toc47510781)

[Remplacer les exceptions avec des tests 17](#_Toc47510782)

[Remplacer une méthode avec un objet de méthode : 18](#_Toc47510783)

[Remplacement de conditions avec clauses gardées 19](#_Toc47510784)

[Utilisation de méthode de requête à la place d’une variable temporaire 20](#_Toc47510785)

[Après : 20](#_Toc47510786)

[Utilisation de champs encapsulés 22](#_Toc47510787)

[} 22](#_Toc47510788)

[Inline methodes 22](#_Toc47510789)

[Design pattern State 22](#_Toc47510790)

[Introduce null object 23](#_Toc47510791)

[Utiliser les stream – Lambda expression 25](#_Toc47510792)

[II. Mise en application 26](#_Toc47510793)

[1. Filter 26](#_Toc47510794)

[Apres : 26](#_Toc47510795)

[2. Comparator 26](#_Toc47510796)

[Apres 26](#_Toc47510797)

[3. .map 26](#_Toc47510798)

[Avant : 26](#_Toc47510799)

[Après : 27](#_Toc47510800)

[4. FindFirst 28](#_Toc47510801)

[Après 28](#_Toc47510802)

[5. ifPresent 28](#_Toc47510803)

[6. FindAny 29](#_Toc47510804)

[Après 29](#_Toc47510805)

[7. FlatMap 29](#_Toc47510806)

[Après : 29](#_Toc47510807)

[8. Utilisation Interface Map, objet HashMap 30](#_Toc47510808)

[9. Processus d’utilisation flatMap 30](#_Toc47510809)

[10. Description chaine de stream API avec map/flatmap/filter/collect 30](#_Toc47510810)

[11. .map avec {} 31](#_Toc47510811)

[12. Description : 31](#_Toc47510812)

[TreeSet<E> 31](#_Toc47510813)

[Travail en cours MGA 33](#_Toc47510814)

[**Les Side effects** 34](#_Toc47510815)

[Parcours objet HashMap, traitement .map et exécution d’opérations complexes 35](#_Toc47510816)

[Stream.of 35](#_Toc47510817)

[.max 35](#_Toc47510818)

[.min 35](#_Toc47510819)

[CompletableFuture 37](#_Toc47510820)

[Mse en application – repository Weibeld 38](#_Toc47510821)

[Mise en application Projet MGA 39](#_Toc47510822)

[Parcours d’une liste de type SOAPMessage 41](#_Toc47510823)

[Webservice REST 44](#_Toc47510824)

[Docker/Kubernates 50](#_Toc47510825)

[Template de création de micro services : 56](#_Toc47510826)

[Avec Spring Boot 56](#_Toc47510827)

[Avec Spring sans SpringBoot 59](#_Toc47510828)

[Cours JAVA 61](#_Toc47510829)

[Collections 63](#_Toc47510830)

[Functional interface 63](#_Toc47510831)

[Non-Static Nested Classes / Inner class 65](#_Toc47510832)

[Static Nested Class 65](#_Toc47510833)

[Différence entre Static Nested Class / Non static Nested class (Inner Class) 65](#_Toc47510834)

[Local class 66](#_Toc47510835)

[Static Method 69](#_Toc47510836)

[Default Method 70](#_Toc47510837)

[Static final / Static 72](#_Toc47510838)

[Static Method 72](#_Toc47510839)

[Autoboxing / Unboxing 73](#_Toc47510840)

[Différences entre JVM JRE JDK 74](#_Toc47510841)

[JVM : 74](#_Toc47510842)

[**Garbage Collector** 74](#_Toc47510843)

[JRE : 74](#_Toc47510844)

[Types d’espaces de stockage 75](#_Toc47510845)

[Java interview questions 75](#_Toc47510846)

[https://www.javacodegeeks.com/java-interview-questions.html 75](#_Toc47510847)

[Thread 76](#_Toc47510848)

[Mémoire globale 76](#_Toc47510849)

[Stack Memory 76](#_Toc47510850)

[Heap Memory 76](#_Toc47510851)

[Abstract class / Interfaces 77](#_Toc47510852)

[Programmation concurrent 79](#_Toc47510853)

[Bloc synchronized 79](#_Toc47510854)

[Design Patterns 80](#_Toc47510855)

[Dao Design Patterns 80](#_Toc47510856)

[Factory Design Pattern Projet MGA 81](#_Toc47510857)

[DAO Design Pattern Projet MGA 81](#_Toc47510858)

[Creational Pattern 83](#_Toc47510859)

[F 83](#_Toc47510860)

[Singleton Design Pattern 87](#_Toc47510861)

[Strategy Design Pattern 91](#_Toc47510862)

[Strategy Design Pattern avec Java 8 95](#_Toc47510863)

[Thread Safe 96](#_Toc47510864)

[Instanciation single thread 96](#_Toc47510865)

[Créer la variable d’instance au temps de chargement de la page. 96](#_Toc47510866)

[Optimisation de la synchronisation 97](#_Toc47510867)

[Utiliser le bloc synchronized à l'intérieur de la boucle conditionnée et avec la variable volatile 97](#_Toc47510868)

[**Behavioural design pattern** 99](#_Toc47510869)

[**Design Pattern State** 99](#_Toc47510870)

[**Cracking the code interview** 100](#_Toc47510871)

[**Définition : Hash Map** 100](#_Toc47510872)

[Si le choix de ne pas redéfinir equals est retenu, l'implémentation d'Object ne fait que comparer les adresses, soit le même comportement que ==. 101](#_Toc47510873)

[**Définition : Hash Table** 101](#_Toc47510874)

[**Définition : Array Lists** 101](#_Toc47510875)

[**Définition : Array String** 101](#_Toc47510876)

[} 101](#_Toc47510877)

[**Définition : String Builder** 102](#_Toc47510878)

[Arrays and Strings : Probleme  : String avec characters uniques 102](#_Toc47510879)

[Array and Strings : Probleme : Permutation d’une String 103](#_Toc47510880)

[Arrays and Strings : Probleme : Urlify 104](#_Toc47510881)

[Array and Strings : Probleme : Permutation d’un palindrome 105](#_Toc47510882)

[LinkedLists 108](#_Toc47510883)

[Création et suppression d’une linked list 108](#_Toc47510884)

[Enlever les duplications 109](#_Toc47510885)

[Partitionner une linkedLIst 110](#_Toc47510886)

[**Threads and Locks** 112](#_Toc47510887)

[Problème 1 : Les philosophes 112](#_Toc47510888)

[Probleme 2 : Les semaphores 116](#_Toc47510889)

[Problème 3 : TODO Synchronized methods 118](#_Toc47510890)

[**Searching and sorting** 119](#_Toc47510891)

[Algorithmes de tri 120](#_Toc47510892)

[Algorithme de recherche 125](#_Toc47510893)

[Binary Search 125](#_Toc47510894)

[**Problèmes** 126](#_Toc47510895)

[Q1 : Sorted Merge 126](#_Toc47510896)

[Q2 : Search Rotated Array 127](#_Toc47510897)

[Spare search 130](#_Toc47510898)

[Sort big file 132](#_Toc47510899)

[Rank from stream 133](#_Toc47510900)

[Find Duplicates 136](#_Toc47510901)

[Pics et Vallées 138](#_Toc47510902)

[Notes sur refactoring été 2020 : 140](#_Toc47510903)

## Méthode de refactoring

### Renommer les variables

Utiliser un nommage adapté pour les variables, caractérisant l’aspect unitaire fonctionnel des méthodes

### Dissocier les boucles for utilisées dans la même méthode

La dissociation permet une utilisation unitaire de la méthode sans complexité

Avant :

public double getGoalsAgainstAverage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

List<Game> games = season.getGames();

int tga = 0;

double mins = 0;

for (Game game: games) {

tga += game.getGoalsAgainst();

mins += game.getMinutesPlayed();

}

return (tga / mins) \* 60;

}

}

public double getSavePercentage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

List<Game> games = season.getGames();

int g = 0;

int tsoga = 0;

for (Game game: games) {

g += game.getGoalsAgainst();

tsoga += game.getShotsOnGoalAgainst();

}

return ((double) tsoga - g) / tsoga;

}

}

### Après :

public double getGoalsAgainstAverage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

List<Game> games = season.getGames();

int tga = 0;

for (Game game: games) {

tga += game.getGoalsAgainst();

}

double mins = 0;

for (Game game: games) {

mins += game.getMinutesPlayed();

}

return (tga / mins) \* 60;

}

}

public double getSavePercentage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

List<Game> games = season.getGames();

int g = 0;

for (Game game: games) {

g += game.getGoalsAgainst();

}

int tsoga = 0;

for (Game game: games) {

tsoga += game.getShotsOnGoalAgainst();

}

return ((double) tsoga - g) / tsoga;

}

}

### Méthode d’extraction des logiques communes présentes dans différentes méthodes

Ceci permet de faire une factorisation du développement.

### Avant

public double getGoalsAgainstAverage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

List<Game> games = season.getGames();

int tga = 0;

for (Game game: games) {

tga += game.getGoalsAgainst();

}

double mins = 0;

for (Game game: games) {

mins += game.getMinutesPlayed();

}

return (tga / mins) \* 60;

}

}

public double getSavePercentage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

List<Game> games = season.getGames();

int g = 0;

for (Game game: games) {

g += game.getGoalsAgainst();

}

int tsoga = 0;

for (Game game: games) {

tsoga += game.getShotsOnGoalAgainst();

}

return ((double) tsoga - g) / tsoga;

}

}

### Apres

public double getGoalsAgainstAverage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

List<Game> games = season.getGames();

int tga = getTotalGoalsAgainst(games);

double mins = 0;

for (Game game: games) {

mins += game.getMinutesPlayed();

}

return (tga / mins) \* 60;

}

}

private int getTotalGoalsAgainst(List<Game> games) {

int g = 0;

for (Game game: games) {

g += game.getGoalsAgainst();

}

return g;

}

public double getSavePercentage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

List<Game> games = season.getGames();

int g = getTotalGoalsAgainst(games);

int tsoga = 0;

for (Game game: games) {

tsoga += game.getShotsOnGoalAgainst();

}

return ((double) tsoga - g) / tsoga;

}

}

### Transformation des boucles en utilisant la librairie Stream API

Le code est simplfié car le parcours de la boucle tient en une seule ligne. Le code est d’avantage lisible.

### Avant

private int getTotalGoalsAgainst(List<Game> games) {

int g = 0;

for (Game game: games) {

g += game.getGoalsAgainst();

}

return g;

}

### Apres

public int getTotalGoalsAgainst() {

return games.stream().mapToInt(game -> game.getGoalsAgainst()).sum();

### Déplacement de méthode dans une classe dédiée

La méthode getTotalGoalsAgainst() utilise l’objet **game** obtenue à partir de la classe **Season**, on déplace donc la méthode dans la classe **Season.**

### Renommage des variables dû aux méthodes de partours des objets, déplacés dans la classe Season

On obtient la classe Game suite au renommage des variables et à l’utilisation des méthodes Stream API

public class GoalieStatistics {

private final Season season;

public GoalieStatistics(Season season) {

this.season = season;

}

public double getGoalsAgainstAverage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

int totalGoalsAgainst = season.getTotalGoalsAgainst();

double totalMinutesPlayed = season.getTotalMinutesPlayed();

return (totalGoalsAgainst / totalMinutesPlayed) \* 60;

}

}

public double getSavePercentage() {

if (season.getGames().isEmpty()) {

return 0.0;

}

else {

int totalGoalsAgainst = season.getTotalGoalsAgainst();

int totalSogAgainst = season.getTotalShotsOnGoalAgainst();

return ((double) totalSogAgainst - totalGoalsAgainst) / totalSogAgainst;

}

}

}

### Renommage des méthodes booléènes de vérification

La méthode **isEmpty** doit être attribué à la classe **Season** pour le lancement d’une saison

### Avant

Class GoalieStatistics {

…

if (season.getGames().isEmpty()) {

return 0.0;

}

### Apres

Class Season {

…

public boolean hasStarted() {

return !games.isEmpty();

}

Class GoalieStatistics {

…

public double getGoalsAgainstAverage() {

if (!season.hasStarted()) {

return 0.0;

}

else {

int totalGoalsAgainst = season.getTotalGoalsAgainst();

double totalMinutesPlayed = season.getTotalMinutesPlayed();

return (totalGoalsAgainst / totalMinutesPlayed) \* 60;

}

}

public double getSavePercentage() {

if (!season.hasStarted()) {

return 0.0;

}

else {

int totalGoalsAgainst = season.getTotalGoalsAgainst();

int totalSogAgainst = season.getTotalShotsOnGoalAgainst();

return ((double) totalSogAgainst - totalGoalsAgainst) / totalSogAgainst;

}

}

Simplifier le conditionnement

On utilise plus la valeur négative du conditionnement mais la valeur **positive**, il n’y aura plus de **else**

### Avant

public double getGoalsAgainstAverage() {

if (!season.hasStarted()) {

return 0.0;

}

else {

int totalGoalsAgainst = season.getTotalGoalsAgainst();

double totalMinutesPlayed = season.getTotalMinutesPlayed();

return (totalGoalsAgainst / totalMinutesPlayed) \* 60;

}

}

public double getSavePercentage() {

if (!season.hasStarted()) {

return 0.0;

}

else {

int totalGoalsAgainst = season.getTotalGoalsAgainst();

int totalSogAgainst = season.getTotalShotsOnGoalAgainst();

return ((double) totalSogAgainst - totalGoalsAgainst) / totalSogAgainst;

}

}

### Après

public double getGoalsAgainstAverage() {

if (season.hasStarted()) {

int totalGoalsAgainst = season.getTotalGoalsAgainst();

double totalMinutesPlayed = season.getTotalMinutesPlayed();

return (totalGoalsAgainst / totalMinutesPlayed) \* 60;

}

else {

return 0.0;

}

}

public double getSavePercentage() {

if (season.hasStarted()) {

int totalGoalsAgainst = season.getTotalGoalsAgainst();

int totalSogAgainst = season.getTotalShotsOnGoalAgainst();

return ((double) totalSogAgainst - totalGoalsAgainst) / totalSogAgainst;

}

else {

return 0.0;

}

}

## Compléments de refactorisation

### Utiliser le pattern Factory pour initialiser le constructeur

Avant

**class** Employee {

Employee(**int** type) {

**this**.type = type;

}

// ...

}

Après

**class** Employee {

**static** Employee create(**int** type) {

employee = **new** Employee(type);

// do some heavy lifting.

**return** employee;

}

// ...

}

* Chargé une seule fois en mémoire
* Static méthodes permet d’ajouter de la sécurité car non accessibles à partir d’une interface implémentée par la classe.
* Plus d’instanciation de l’actuelle implémentation des classes depuis le code client,

### Remplacer le conditionnel par du polymorphisme

Avant

**class** Handler

{

public void increasePoints(GameObject go1)

{

**if**(**type**Of(go1)== Ship )

{

*//increase points by 5*

}

**else** **if**(**type**Of(go1)==Station)

{

*//increase points by 10*

}

**else** **if**(**type**Of(go1)==Commet)

{

*//increase points by 15*

}

}

}

Apres

Interface GameObject(){

increasePoints(points){};

}

**class** Ship implements GameObject {

increasePoints(points){

*//increase points by 5*

}

}

**class** Station implements GameObject {

increasePoints(points){

*//increase points by 10*

}

}

**class** Commet implements GameObject {

increasePoints(points){

*//increase points by 15*

}

}

**class** Handler {

**public** void increasePoints(GameObject go1) {

go1.increasePoints(points);

}

}

Pour appeler la classe abstraite, permettant de créer une liste de sous classes et réaliser un traitement specifique à chaque sous classes, utiliser le code suivant :

*Note : on déclare un objet de type Classe Abstraite, et on invoque les méthodes des sous classes étandant la classe abstraite.*

**public** **class** Handler **extends** TraitementTypeColisAbstract{

@Override

**public** CreationColisVO traitement(CreationColisVO creationColisVO) {

// **TODO** Auto-generated method stub

**return** **null**;

}

**public** **void** perform(CreationColisVO creationColisVO) {

List<TraitementTypeColisAbstract> tlistAbstract = **new** ArrayList<TraitementTypeColisAbstract>() {{

add(**new** TraitementTypeColisVide());

add(**new** TraitementTypeColisNonVide());

}};

**for**(TraitementTypeColisAbstract traitementTypeColisAbstract : tlistAbstract) {

traitementTypeColisAbstract.traitement(creationColisVO);

}

//Utiliser le design pattern factory pour instancier les sous classes de la classes abstraite

}

### Supprimer les assignements de paramètres

Avant

**int** discount(**int** inputVal, **int** quantity) {

**if** (inputVal > 50) {

inputVal -= 2;

}

// ...

}

Apres

**int** discount(**int** inputVal, **int** quantity) {

int result = inputVal ;

**if** (result > 50) {

result = -2

}

// ...

}

### Remplacer les exceptions avec des tests

Avant

**double** getValueForPeriod(**int** periodNumber) {

**try** {

**return** values[periodNumber];

} **catch** (ArrayIndexOutOfBoundsException e) {

**return** 0;

}

}

Après

**double** getValueForPeriod(**int** periodNumber) {

**if** (periodNumber >= values.length) {

**return** 0;

}

**return** values[periodNumber];

}

### Remplacer une méthode avec un objet de méthode :

Avant :

**class** Order {

// ...

**public** **double** price() {

**double** primaryBasePrice;

**double** secondaryBasePrice;

**double** tertiaryBasePrice;

// Perform long computation.

}

}

Apres :

**class** Order {

// ...

**public** **double** price() {

**return** **new** PriceCalculator(**this**).compute();

}

}

**class** PriceCalculator {

**private** **double** primaryBasePrice;

**private** **double** secondaryBasePrice;

**private** **double** tertiaryBasePrice;

**public** PriceCalculator(Order order) {

// Copy relevant information from the

// order object.

}

**public** **double** compute() {

// Perform long computation.

}

}

### Remplacement de conditions avec clauses gardées

Avant

**public** **double** getPayAmount() {

**double** result;

**if** (isDead){

result = deadAmount();

}

**else** {

**if** (isSeparated){

result = separatedAmount();

}

**else** {

**if** (isRetired){

result = retiredAmount();

}

**else**{

result = normalPayAmount();

}

}

}

**return** result;

}

Après

**public** **double** getPayAmount() {

**if** (isDead){

**return** deadAmount();

}

**if** (isSeparated){

**return** separatedAmount();

}

**if** (isRetired){

**return** retiredAmount();

}

**return** normalPayAmount();

}

### Utilisation de méthode de requête à la place d’une variable temporaire

Au lieu d’utiliser une variable temporaire :

Avant :

String monAtelier = ‘’

**if**(StringUtils.isNotBlank(redditionVO.getNumAtelierLibelle()) && redditionVO.getNumAtelierLibelle().contains("-")){

monAtelier = StringUtils.trim(StringUtils.split(redditionVO.getNumAtelierLibelle() , "-")[0]);

}

### Après :

**if**(StringUtils.isNotBlank(redditionVO.getNumAtelierLibelle()) && redditionVO.getNumAtelierLibelle().contains("-")){

//AHO - Apres - refactoring

monAtelier = getAtelier(redditionVO);

}

**public** String getAtelier(RedditionVO redditionVO) {

**if**(StringUtils.isNotBlank(redditionVO.getNumAtelierLibelle()) &&

redditionVO.getNumAtelierLibelle().contains("-")){

**return** StringUtils.trim(StringUtils.split(redditionVO.getNumAtelierLibelle() , "-")[0]);

}

**return** **null**;

}

Remplacement conditionnement complexe par une methode regroupant toutes les conditions :

**if** (date.before(SUMMER\_START) || date.after(SUMMER\_END)) {

charge = quantity \* winterRate + winterServiceCharge;

}

**else** {

charge = quantity \* summerRate;

}

Par

**if** (isSummer(date)) {

charge = summerCharge(quantity);

}

**else** {

charge = winterCharge(quantity);

}

### Utilisation de champs encapsulés

Avant

**class** Range {

**private** **int** low, high;

**boolean** includes(**int** arg) {

**return** arg >= low && arg <= high;

}

}

Après

**class** Range {

**private** **int** low, high;

**boolean** includes(**int** arg) {

**return** arg >= getLow() && arg <= getHigh();

}

**int** getLow() {

**return** low;

}

**int** getHigh() {

**return** high;

}

### }

### Inline methodes

**class** **PizzaDelivery** {

*// ...*

**int** getRating() {

**return** moreThanFiveLateDeliveries() ? 2 : 1;

}

**boolean** moreThanFiveLateDeliveries() {

**return** numberOfLateDeliveries > 5;

}

}

### Design pattern State

Pour traitement de méthodes trop longues :

Utiliser :

* Une classe abstraite (**EtapesAbstract**) avec les champs utilisés par les sous classes étendant cette classe
* Des sous classes (**ConnectDatabase**, **InitializeSortDate**) réalisant des traitements unitaires (traitement déclaré dans la classe abstraite en tant que méthode abstraite).
* Des méthodes **prev** et **next** sont utilisés pour les chainage des traitements
* Une classe basique permet la définition de l’objet classe abstraite pour réaliser la suite des traitements (**setState, setPrevious, setNext**)

Exemple d’implémentation :

Etape etape = **new** Etape();

etape.setState(**new** Etape1());

etape.getState().treatment();

etape.setNext();

etape.setState(**new** Etape2());

etape.getState().treatment();

Classe basique (**Etape**)

**public** **class** Etape {

**private** EtapesAbstract state = **new** Etape1();

**public** **void** setState(EtapesAbstract state) {

**this**.state = state;

}

**public** **void** setPrevious() {

getState().previous(**this**);

}

**public** **void** setNext() {

getState().next(**this**);

}

**public** EtapesAbstract getState() {

**return** state;

}

}

### Introduce null object

Remplacement des valeurs nulles par objets nuls

**class** **NullCustomer** **extends** Customer {

**boolean** isNull() {

**return** **true**;

}

Plan getPlan() {

**return** **new** NullPlan();

}

*// Some other NULL functionality.*

}

*// Replace null values with Null-object.*

customer = (order.customer != **null**) ?

order.customer : **new** NullCustomer();

*// Use Null-object as if it's normal subclass.*

plan = customer.getPlan();

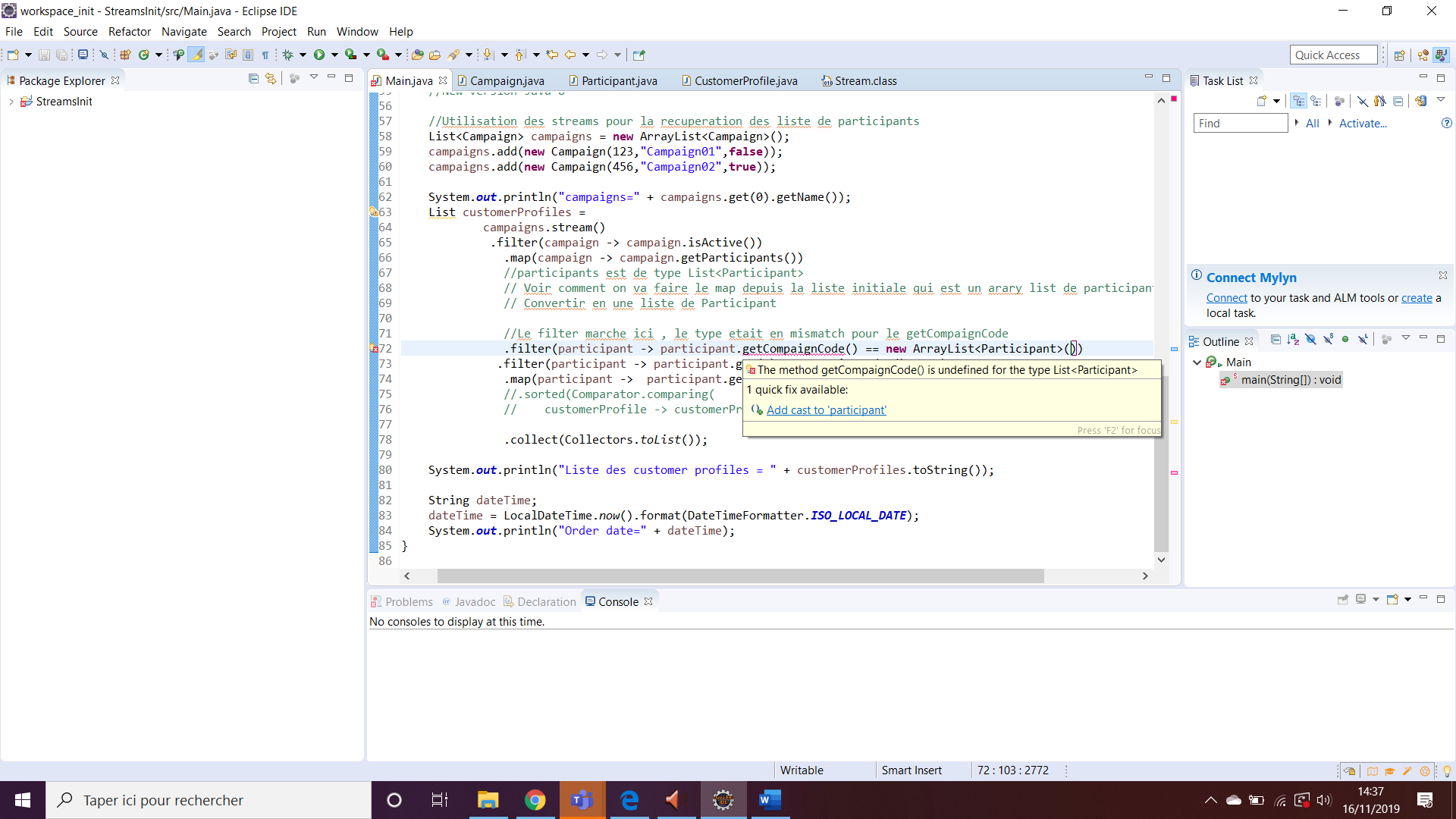
# Utiliser les stream – Lambda expression

Formation avec <https://www.exclamationlabs.com/blog/refactoring-for-java-8-streams/>

OK - Utilisation des streams

Problème rencontrés :

Impossible de faire le filter sur la liste des participants pour filtrer sur le campaign code == 1 , le return type de getCompaignCode doit être de List<Participant> ce qui est le cas.



Il est possible de créer un .map a partir d’une campaign pour retourner une liste de participants. Pas de probleme d’externalisation des objets dans une classe externe. Le flux de map pour prendre en compte une liste d’objets.

Interfaces fonctionnelles :

<https://www.oreilly.com/learning/java-8-functional-interfaces>

## II. Mise en application

### Filter

Méthode filter pour réaliser conditionnement : if classique

Attention :

Combiner deux conditions avec l’opérateur **&&** et **||**

### Apres :

List<EstActiveEntity> historized = results.stream()

.filter(eac -> eac.getHisto() != **null** && EtatActiveEnum.***HISTORISEE***.getCode().equals(eac.getEtaId()))

.sorted((x, y) -> x.getHisto().compareTo(y.getHisto())).collect(Collectors.*toList*());

### Comparator

Comparer deux éléments dans un flux stream

On peut utiliser Comparator.comparing() :

For example, to **create a comparator**, the following syntax is enough:

|  |  |
| --- | --- |
| 1 | Comparator c = (Computer c1, Computer c2) -> c1.getAge().compareTo(c2.getAge()); |

Then, with type inference:

|  |  |
| --- | --- |
| 1 | Comparator c = (c1, c2) -> c1.getAge().compareTo(c2.getAge()); |

But can we make the code above even more expressive and readable? Let's have a look:

|  |  |
| --- | --- |
| 1 | Comparator c = Comparator.comparing(Computer::getAge); |

We've used the *::* operator as shorthand for lambdas calling a specific method – by name. And the end, result is of course even more readable syntax.

### Apres

Date dateDebutMandat = prestaOperateurDto.getPrestaImplantations()

.stream()

.map(x -> x.getDateDebut())

.filter(x -> x != **null**)

.min((d1, d2) -> d1.compareTo(d2))

.get();

### .map

Modifie le type de retour du flux Stream.

*Attention :* après un **map** pour retourner une Liste, il faut utiliser un **.collect(Collectors.toList())** ou **.collect(Collectors.toSet())** pour ensuite pouvoir utiliser de nouveau le flux transformé.

Ceci permet de réaliser en mode Java 8, les enchainements réalisés en java

7.

### Avant :

List etatControleFinale = **new** ArrayList();

Iterator it = listeEtatsControle.iterator();

Map etatControleMap = **new** HashMap();

**while** (it.hasNext()) {

IEtatControleNaeBO etatControleNae = (IEtatControleNaeBO) it.next();

String cleMapFix = etatControleNae.getUa()+etatControleNae.getLigne();

**if** ( (etatControleMap.get(cleMapFix + etatControleNae.getEs1coupure()) != **null**)

|| (etatControleMap.get(cleMapFix + CoupureUtil.*convertirCoupureES2enES1*(etatControleNae.getEs2coupure()) ) != **null**)){

### Après :

List etatControleFinale = **new** ArrayList();

Map etatControleMap = **new** HashMap();

List<IEtatControleNaeBO> listeEtatsControleNaeBO = **new** ArrayList<IEtatControleNaeBO>();

Set<String> **newListEtatsControle\_step0** = (Set) listeEtatsControleNaeBO

.stream()

.map(etatControleNae -> (String) (((IEtatControleNaeBO) etatControleNae).getUa()) + (String)(((IEtatControleNaeBO) etatControleNae).getLigne()))

.collect(Collectors.*toSet*());

// et ensuite introduire le if

//if ( (etatControleMap.get(cleMapFix + etatControleNae.getEs1coupure()) != null)

**for**(String etatControle : **newListEtatsControle\_step0**)

{

listeEtatsControleNaeBO.stream()

//Condition remplacant

/\*

\* if ( (etatControleMap.get(cleMapFix + etatControleNae.getEs1coupure()) != null)

|| (etatControleMap.get(cleMapFix + CoupureUtil.convertirCoupureES2enES1(etatControleNae.getEs2coupure()) ) != null)){

\*/

.filter(

(etatControleStream2 -> (etatControleMap.get(etatControle + etatControleStream2.getEs1coupure()) != **null**))

)

//Condition remplacant

/\*

\* if ( etatControleNae.getModeTraitement() == 4 || etatControleNae.getModeTraitement() == 5){

\*/

.filter(

etatControleStream2 -> (etatControleStream2.getModeTraitement() == 3) || (etatControleStream2.getModeTraitement() == 4)

)

.forEach(etatControleStream2 -> etatControleStream2.setEs2valides(1))

;

}

*Attention :* Il est possible de réaliser un traitement complexe dans un .map (utilisation des **crochets {}** avec valeur de retour correspondant à l’input du .map)

List<APourContact> nationaux = getRepository().findNationauxByOperateur(id).stream()

.map(aPourContactMapper::toModel).map(apc -> {

apc.setContact(contactMapper.toModel(contactRepository.findById(apc.getId().getIdContact())));

**return** apc;

})

.collect(Collectors.*toList*());

Long countUnclosedAnomalies = golfiTableErrEntities.stream().map(e -> {

StringBuilder jpql = **new** StringBuilder();

jpql.append("select count(g.id)");

jpql.append(" from ").append(e).append(" g");

jpql.append(" where g.statut = :statut");

TypedQuery<Long> query = getEntityManager().createQuery(jpql.toString(), Long.**class**);

query.setParameter("statut", Integer.*valueOf*(0));

**return** query.getSingleResult();

}).mapToLong(res -> res).sum();

### FindFirst

Permet de retourner un container de type **Optional** **null ou non null**. Celui-ci retourne le premier element du stream trouvé non null. Si le stream est vide, il retourne un Container optional de type null

Dans le cas du projet GOLFI, on effectue un filtre sur le stream et on récupère le premier élement trouvé dont l’id est égal à l’id de l’objet

### Après

**for** (EstActive newEac : newEacs) {

oldEacs.stream().filter(oldEac -> oldEac.getActId().equals(newEac.getActId())).findFirst()

.ifPresent(oldEac -> {

String typeAct = activiteRepository.findLibelleTypeActByActId(newEac.getActId());

tracesModificationActivite

.addAll(TraceModificationUtil.*getImpModificationDatesEACMsgs*(oldEac, newEac, typeAct));

});

}

### ifPresent

Après un **findFirst**, on utilise la méthode **ifPresent** pour réaliser un traitement complexe entre accolades **{}**

**for** (EstActive newEac : newEacs) {

oldEacs.stream().filter(oldEac -> oldEac.getActId().equals(newEac.getActId())).findFirst()

.ifPresent(oldEac -> {

String typeAct = activiteRepository.findLibelleTypeActByActId(newEac.getActId());

tracesModificationActivite

.addAll(TraceModificationUtil.*getImpModificationDatesEACMsgs*(oldEac, newEac, typeAct));

});

}

### FindAny

Retourne un Optional<>

findAny() is a ***terminal-short-circuiting*** operation of Stream interface. This method returns any first element satisfying the intermediate operations. This is a short-circuit operation because it just needs **‘any’** first element to be returned and terminate the rest of the iteration.

If the optional is empty, an empty Optional<> is returned

### Après

ACTBPSectionAggregate aggregate = (ACTBPSectionAggregate) sectionIdGroup.stream().findAny().get();

### FlatMap

La méthode flatMap permet de mettre a plat les donnes regroupées sous fore de couple ou d’organisation

Le résultat de la méthode flatmap est une objet de type List

### Après :

Map<String, List<Section>> formSections = rm.convertToSections(remise);

state.setFormSections(formSections);

List<String> reportSections = formSections.values().stream().flatMap(List::stream).map(Section::getSectionId)

### Utilisation Interface Map, objet HashMap

* **.values()**

***values****()****method****of HashMap class in****Java****is used to create a collection out of the****values****of the map. It basically returns a Collection view of the****values****in the HashMap. Parameters: The****method****does not accept any parameters.*

Map<String, List<Section>> formSections = rm.convertToSections(remise);

List<String> reportSections = formSections.values().stream().flatMap(List::stream).map(Section::getSectionId)

.collect(Collectors.*toList*());

**Initialiser une HashMap**

    articleMapOne = new HashMap<>();

    articleMapOne.put("ar01", "Intro to Map");

    articleMapOne.put("ar02", "Some article");

### Processus d’utilisation flatMap

La méthode **flatmap** retourne un objet de type Stream<Object> à partir d’un objet Steam<Section> sous forme de List

Stream<List<Section>> reportSections2 = formSections2.values().stream().flatMap(List::stream);

### Description chaine de stream API avec map/flatmap/filter/collect

Description de la chaine globale :

Set<ConventionEntity> oldConventions = oldActivites

.stream()

.map(act -> act.getConventions())

.flatMap(

cnvs -> cnvs.stream()

)

.filter(

cnv -> (

cnv.getId() != **null** && cnv.getHisto() == **null**

)

)

.collect(

Collectors.*toCollection*(

() -> **new** TreeSet<>((c1, c2) -> (**int**) (c1.getId() - c2.getId()))));

* La classe Set<ConventionEntity> est une **interface** de type **Set**

### .map avec {}

La méthode map peut être utilisée avec {} pour utiliser des propriétés des attributs du stream

String a="";

map.entrySet()

.stream()

.map(m -> {

Stream<Object> test = m.getValue()

.stream()

.map(t -> t.toString());

a.toString();

**return** m.getValue().stream();

}

);

/ !\ : Ne pas oublier le return à l’intérieur des {}

/ !\ : La variable a ne peut être setté en dehors du stream, sinon celle-ci n’est plus effectively final

### Description :

On utilise l’interface **TreeSet** pour pouvoir trier les collections :

### TreeSet<E>

:

Set<ConventionEntity> oldConventions = oldActivites.stream().map(act -> act.getConventions())

.flatMap(

cnvs -> cnvs.stream()

)

* Remise à plat sous forme de **Stream<ConventionEntity>**

.filter(cnv -> (cnv.getId() != **null** && cnv.getHisto() == **null**))

* Filtre sur la présence d’un identifiant non null et d’un historique non null

.collect

(

Collectors.*toCollection*

(

() -> **new** TreeSet<>

(

(c1, c2) -> (**int**) (c1.getId() - c2.getId())

)

)

);

* Après la méthode .collect, l’objet est convertit sous forme de Set<ConventionEntity>, objet de départ est instancié avec l’interface **TreeSet** qui permet de récupérer un ordre dans l’arbre

## Travail en cours MGA

*Fichier : MainClass.java*

*D:\DEV\workspace\pjbv2\_oracle12\_v3\sport-batchs\sport-exportInfo2\src\main\java\fr\bdf\sport\info2\batch\MainClass.java*

// ---------------------------------- récupération des sessions MGA pour Info2 -------------------

Compléter les fonctionnalités utilisées à partir du code modifié java 8

* **Utilisation méthode filter**

//Déclaration du stream principal

Stream<ISession4Info2BO> filteredISessionInfo2BOStream = sessionsMGA\_java8.stream()

//.filter((Objects::nonNull))

.filter(conditionNotNull.or(conditionGetIdentifiant).and(conditionNotNull))

;

Stream<ISession4Info2BO> mapSessionsTraites\_java8IsNull = filteredISessionInfo2BOStream

.filter(x -> x == **null**);

Stream<ISession4Info2BO> dernierIdSession\_java8IsNotNull = filteredISessionInfo2BOStream

.filter(x -> dernierIdSession\_java8 != **null**);

**Utilisation méthode map**

Permet de transformer un type d’elements du stream en un autre

Nécessaire de caster le stream utilisé en type de la variable utilisé

Exemple : **aggregate** est de type **List<DMOPSectionAggregate>**

Un .map est ensuite fait en sortie du flux e

Un .collect est ensuite fait pour transformer le **stream** en **list**

List<SectionAggregate> aggs = (List<SectionAggregate>) candidates[1];

List<DMOPSectionAggregate> aggregates = (List<DMOPSectionAggregate>) aggs.stream()

.map(e -> (DMOPSectionAggregate) e).collect(Collectors.toList());

**Attention :**

Ne pas utiliser la méthode **.map** combiné avec la méthode **forEach**

L’erreur retournée dans ce cas est la suivante :

Cannot cast from void to List<TypeDeClasse>

Ou

Cannot invoke map() on the primitive type void

# **Les Side effects**

Un Side effect est une action effectuée par une opération d’un stream modifiant quelque chose d’externe. Par exemple, modifier une variable de l’application, envoyer un message JMS, envoyer un email, modifier l’interface graphique dans une application, etc.

Toutes les opérations ne sont pas autorisées à utiliser des side effects.

Par exemple :

* ForEach(), ForEachOrdered() et peek() renvoient void et sont donc destinées à produire un side effect.
* Les opérations intermédiaires avec des paramètres comportementaux de type map, filter, etc.. qui ne retournent pas de valeur void ne devraient pas utiliser des side effects.



**Attention :**

Le traitement **forEach** doit utiliser l’expression lambda déclarée initialement



Pas de méthode map combiné avec méthode ForEach (sideEffect)

Car la méthode forEach retourne void et map / filter retournent généralement une valeur.



//Provoque erreur Local variable mapSessionsTraites\_java8 defined in an enclosing scope must be final or effectively final

dernierIdSession\_java8IsNotNull.forEach(i -> mapSessionsTraites\_java8 = **new** HashMap<Integer, List<ISession4Info2BO>>());

Local variable mapSessionsTraites\_java8 defined in an enclsosing scope must be final or effectively final

## Parcours objet HashMap, traitement .map et exécution d’opérations complexes

Pour réaliser des opérations concrètes après un traitement .map (exemple consolidation de données, traitement quelconque), il est nécessaire de reconvertir l’objet Stream en List :

La liste recrée contient les éléments qui ont été mappés dans le 1er stream

1. Parcourir la liste ainsi crée
2. Effectuer un 2ème stream de l’objet List<> initial
3. Réaliser des conditions avec la méthode **filter**
4. Effectuer un traitement avec la méthode **forEach**

**private** List traiterEtatControleParCoupure(List listeEtatsControle) {

//Java 8

List etatControleFinale = **new** ArrayList();

Map etatControleMap = **new** HashMap();

List<IEtatControleNaeBO> listeEtatsControleNaeBO = **new** ArrayList<IEtatControleNaeBO>();

**Etape 1**

Set<String> newListEtatsControle\_step0 = (Set) listeEtatsControleNaeBO

.stream()

.map(etatControleNae -> (String) (((IEtatControleNaeBO) etatControleNae).getUa()) + (String)(((IEtatControleNaeBO) etatControleNae).getLigne()))

.collect(Collectors.*toSet*());

**Etape 2 A modifier en imbriquant le 2eme stream dans le 1er (cf ComparableFuture)**

**for**(String etatControle : newListEtatsControle\_step0)

{

listeEtatsControleNaeBO.stream()

**Etape 3** .filter(

(etatControleStream2 -> (etatControleMap.get(etatControle + etatControleStream2.getEs1coupure()) != **null**) || **true**)

)

**Etape 4** .forEach(etatControleStream2 -> etatControleStream2.setEs2valides(1));

}

### Stream.of

Permet de construire un Stream à partir d’elements pour ensuite effectuer des comparaisons

### .max

Permet d’effectuer des comparaisons et de récupérer le maximum entre plusieurs éléments

### .min

* Permet d’effectuer des comparaisons et de récupérer le minimum entre plusieurs éléments

Long minFlux = Stream.*of*(qtPresumeFaux, qtTraitee, qtRecu, qtRemisEnCercul).min((x, y) -> x.compareTo(y))

Date dateDebutMandat = prestaOperateurDto.getPrestaImplantations().stream()

.map(x -> x.getDateDebut()).filter(x -> x != **null**)

.min((d1, d2) -> d1.compareTo(d2))

.get();

**.**sorted

Permet de trier une liste

List<EstActiveEntity> historized = results.stream()

.filter(eac -> eac.getHisto() != **null** && EtatActiveEnum.***HISTORISEE***.getCode().equals(eac.getEtaId()))

.sorted((x, y) -> x.getHisto().compareTo(y.getHisto())).collect(Collectors.*toList*());

## CompletableFuture

**Documentation pour MEA** : <https://medium.com/@kalpads/fantastic-completablefuture-allof-and-how-to-handle-errors-27e8a97144a0>

**Definition :**

Classe d’une interface fonctionnelle implémentant l’interface Future

Description de CompletableFuture :

The *Future* interface was added in Java 5 to serve as a result of an asynchronous computation, but it did not have any methods to combine these computations or handle possible errors.

**In Java 8, the *CompletableFuture* class was introduced.** Along with the *Future* interface, it also implemented the *CompletionStage* interface. This interface defines the contract for an asynchronous computation step that can be combined with other steps.

Cf <https://www.baeldung.com/java-completablefuture>

<https://nickebbitt.github.io/blog/2017/03/22/async-web-service-using-completable-future>

**Méthode useAsync**

he CompletableFuture.supplyAsync run asynchronously a Supplier functional interface which represents by the lambda expression () -> "Hello future!"

The completableFuture.get() blocks until the completableFuture is complete and return the result

Apart from using get(), you can also use its result to continuously execute another methods in the callbacks chain

<https://hellokoding.com/java-8-completablefuture-supplyasync-tutorial-with-examples/>

**Fil rouge pour description fonctionnalité :**

CompletableFuture<Void> task =

CompletableFuture.*allOf*(

Stream.*of*(Product.Category.*values*())

.map(

category -> *supplyAsync*(

() -> productRepository.productsByCategory(category),executor

)

.thenAccept(

products -> productsByCategories.put(category,products)

)

)

.toArray(CompletableFuture[]::**new**));

### Mse en application – repository Weibeld

<https://github.com/weibeld/Java-8-CompletableFuture-Demo>

Creation du workspace contenant le repository

<https://medium.com/@kalpads/fantastic-completablefuture-allof-and-how-to-handle-errors-27e8a97144a0>

**2eme version** :

* Methode **supplyAsync** pour gérer les méthodes du stream de manière asynchrone
* Suppression du cast en **CompletableFuture**< ?> en amont du traitement, on utilise directement un **Collectors.toArray(CompletableFuture[]**) en aval pour ne plus caster

CompletableFuture<Void> tasks4 = CompletableFuture.*allOf*(

listeEtatsControleNaeBO.stream()

.map(etatControleNaeBo -> *supplyAsync*(

()-> etatControleNaeBo.getUa(),executor))

.toArray(CompletableFuture[]::**new**));

**3eme version :**

* Méthode **allOf** pour attente de la réalisation de tousl es completableFutures

CompletableFuture<Void> tasks4 = CompletableFuture.*allOf*(

listeEtatsControleNaeBO.stream()

.map(etatControleNaeBo -> *supplyAsync*(

()-> etatControleNaeBo.getUa(),executor))

.toArray(CompletableFuture[]::**new**));

CompletableFuture<Void> allFutures = CompletableFuture

.*allOf*(((List<IEtatControleNaeBO>) completableFutures).toArray(**new** CompletableFuture[((List) completableFutures).size()]));

### Mise en application Projet MGA

1. Récupération des billets pour un résultat de tri de manière asynchrone
   1. Utilisation d’une hashMap et parcours de la liste de valeur
   2. Utilisation de la méthode **getNbBillets\_async** de manière asynchrone

Map<String, List<ResultatTriBO>> mapResTrisBySSCC2 = **new** HashMap<String, List<ResultatTriBO>>();

mapResTrisBySSCC2 = getResultatTriBS().trouverResultatTrisBySSCCOrigine(recDeMeC, coupure.getValeurFaciale(), codeCtgBillet);

List<List<CompletableFuture<Integer>>> listListNbBillets = mapResTrisBySSCC2.entrySet()

.stream()

.filter(f0 -> !f0.equals(ColisBO.***SSCC\_INCONNU***))

.map(Map.Entry::getValue)

.map(m2 -> m2

.stream()

.filter(

resultatTri -> ArticleUtil.*getCodeCoupureByArticle*(resultatTri.getIdGTINArticle())==coupure.instanceEquivalentS1().getCode()

)

.map(m3 -> m3.getNbBillets\_async())

.collect(Collectors.*toList*())

)

.collect(Collectors.*toList*());

* 1. Déclaration méthode **getNbBillets\_async** (**ResultatTriBO.java**)

**public** CompletableFuture<Integer> getNbBillets\_async() {

**return** CompletableFuture.*supplyAsync*( () -> {

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**return** nbBillets;

}, executor);

}

**private** ExecutorService executor = Executors.*newFixedThreadPool*(4);

1. Déclaration d'un **allFutures** vide, enchainant tous les future a être réalisés dans la map

CompletableFuture<Void> allFutures = CompletableFuture

.*allOf*(listListNbBillets.toArray(**new** CompletableFuture[listListNbBillets.size()]));

) ;

1. Réalisation de tous les futures et concatenation dans **allCompletableFuture**

List<List<CompletableFuture<Integer>>> allCompletableFuture = (List<List<CompletableFuture<Integer>>>)

.thenApply(future ->

{

**return** listListNbBillets.stream()

.map(m -> m.stream()

.map(m2 -> m2.join())

collect(Collectors.*toList*()))

.collect(Collectors.*toList*());

});

### Parcours d’une liste de type SOAPMessage

1. **Déclaration de la liste sur laquelle on veut faire un supplyAsync**

  List<FluxMidefBO> listFlux = new ArrayList<FluxMidefBO>();

1. **Déclaration d'un executor**

 private ExecutorService mExec = Executors.newFixedThreadPool(  
Math.min(100,listSoapMessages.size()),  
runnable -> {Thread t = new Thread(runnable);  
t.setDaemon(true);  
return t;}  
);

1. **Déclaration de la méthode asynchrone SOAPMessage**

private CompletableFuture<SOAPMessage> appelWsSoap\_async(String url, FluxMidefBO flux)  
{  
return CompletableFuture.supplyAsync(() -> {  
return appelWsSoap(url,flux);  
},mExec);  
}

1. **Parcours de la liste initiale pour retourner List<CompletableFuture<SOAPMessage>>**

  List<CompletableFuture<SOAPMessage>> allCompletableFutures = listFlux  
.stream()  
.map(fluxMidef ->   
(CompletableFuture<SOAPMessage>) appelWsSoap\_async(getUrlMoges(),fluxMidef) )  
.collect(Collectors.toList());

1. **.allOf pour retourner un objet de type CompletableFuture<Void>**

CompletableFuture<Void> allFutures = CompletableFuture.allOf(  
allCompletableFutures.toArray(new CompletableFuture  
[allCompletableFutures.size()]  
)  
);

1. **thenApply pour terminer le processus et retourner un objet de type List<CompletableFuture<SOAPMessage>>**

List<CompletableFuture<SOAPMessage>> completedFutures = (List<CompletableFuture<SOAPMessage>>) allFutures.thenApply(  
future -> {

return allCompletableFutures  
.stream()  
.map(completedFuture -> completedFuture.join());  
});

Parcours d’une liste de type HashMap avec la fonctionnalité entrySet

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

List<List<FluxMidefBO>> test = map

.entrySet()

.stream()

.map(Map.Entry::getValue)

.map(m -> m.stream()

.map(m2 ->m2)

.collect(Collectors.*toList*())

)

.collect(Collectors.*toList*());



Attention :

1ere etape : Déclaration d’une liste de de CompletableFuture<**Objet**>

2eme etape : Déclaration d’une liste de CompletableFuture<**Void**>

3eme etape : Déclaration d’un objet de type CompletableFuture<**List<Objet>**>

4eme etape : Déclaration d’un objet de type **CompletableFuture**



Ne pas déclarer les getter et setter en mode static

Sinon erreur dans le .map avec **l’utilisation du double semi colon ‘ ::’**

CompletableFuture completableFuture = allCompletableFuture.thenApply(

typeDeProduit -> {

**return** typeDeProduit.stream()

.map(TypeDeProduit::*getProduit*)

.collect(Collectors.*toList*());

}

);

Avec les getter et setter suivants :

**public** **static** String getProduit() {

**return** *typeDeProduit*;

}

Entraine l’erreur suivante :

The method getProduit() from the type TypeDeProduit should be accessed in a static way

### Webservice REST

Suivi Formation <https://mbaron.developpez.com/tutoriels/soa/developpement-services-web-jaxrs-maven-eclipse/#LI>

Git Hub <https://github.com/mickaelbaron/jaxrs-tutorial/tree/solution>

JDK 8

Création d’une façade pour requête de type **POST** et **GET**

Paramètre d’entrée pour les requêtes **POST** peuvent être du **JSON** ou des String

Cette classe correspond à un **controller (**CalculTauxFacade)

Attributs d’entrée utilisés en amont de la classe :

@RequestScoped

@Path("/mga/calculTaux")

@Produces(MediaType.***APPLICATION\_JSON***)

@RolesAllowed({"SPORT\_STATLIMITE","SPORT\_STATLIMITE\_IEDOM","SPORT\_STATCOMPLET"})

**public** **class** CalculTauxFacade {

@POST

@Path("/prepareExport")

@Consumes(MediaType.***APPLICATION\_JSON***)

**public** ExportFichierVO prepareExport (CalculTauxVO calculTauxVO){

User user = SportContextManager.*getUserContext*(httpServletRequest);

**if** (user == **null**)

**return** **null**;

HSSFWorkbook file = prepareExportCalculTauxAction.performReal(calculTauxVO,user);

**return** creerFichierEtResponseExcel(file, calculTauxVO);

}

**Configuration utilisée dans le pom.xml de sport-rest**

<configuration>

<wadlFile>${project.build.directory}/classes/application.wadl</wadlFile>

<formatWadlFile>true</formatWadlFile>

<baseUri>http://127.0.0.1/sport/rest</baseUri>

<packagesResourceConfig>

<params>

<param>fr.bdf.sport.facade</param>

<param>fr.bdf.mga.facade</param>

</params>

</packagesResourceConfig>

<wadlGenerators>

<wadlGeneratorDescription>

<className>com.sun.jersey.server.wadl.generators.WadlGeneratorApplicationDoc

</className>

<properties>

<property>

<name>applicationDocsFile</name>

<value>src/main/doc/application-doc.xml</value>

</property>

</properties>

</wadlGeneratorDescription>

<wadlGeneratorDescription>

<className>com.sun.jersey.server.wadl.generators.WadlGeneratorGrammarsSupport

</className>

<properties>

<property>

<name>grammarsFile</name>

<value>src/main/doc/application-grammars.xml</value>

</property>

</properties>

</wadlGeneratorDescription>

</wadlGenerators>

</configuration>

Sport-rest contient l’ensemble des appels WS de type JAX-RS-WebServices

Formation via <https://mbaron.developpez.com/tutoriels/soa/developpement-services-web-jaxrs-maven-eclipse/#LI>

Utilisation de webservice SOAP pour consommer du XML

Utilisation de webservices REST pour consommer du JSON

* Création d’un rest endpoint en utilisant jax-rs avec l’annotation **@ApplicationPath**

Les facades rest sont déclarées ici :

@ApplicationPath("/rest")

**public** **class** SportRestWebServiceApplication **extends** Application {

**public** Set<Class<?>> getClasses() {

Set<Class<?>> serviceResources = **new** HashSet<Class<?>>();

serviceResources.add(PrpIdentColisFacade.**class**);

* Exposition des services REST avec les endpoint (facades) déclarées en utilisant les annotations **@Path**, **@Produce** et **@RequestScoped**

@RequestScoped

@Path("/mga/preparationTri")

@Produces(MediaType.***APPLICATION\_JSON***)

@RolesAllowed({"SPORT\_ADMINISTRATION","SPORT\_MGA\_PREP"})

* Création d’une classe racine avec un attribut **@Path** sur lequel va être joué la requête

Définition du type de retour (TEXT\_PLAIN)

@Path("hello")

@Produces(MediaType.TEXT\_PLAIN)

**public** class HelloResource {

**public** HelloResource() { }

}

Définition d’une première méthode retournant la réponse :

@GET

**public** String getHello() {

**return** "Bonjour ENSMA";

}

Ajout des dépendances JAX-RS dans le pom.xml du projet

<dependency>

<groupId>javax.ws.rs</groupId>

<artifactId>javax.ws.rs-api</artifactId>

<version>${jaxrs.version}</version>

</dependency>

II.

Ajout d’une méthode variabilisant le message avec l’attribut **@PathParam(«id »)** en paramètre de la méthode, et **@Path(« {id} »)** en attribut de la méthode :

@GET

@Path("{id}")

**public** String getHello(@PathParam("id") String id,

@DefaultValue("votre serviteur") @HeaderParam("name") String name) {

**return** "Bonjour " + id + " de la part de " + name;

}

Appel de la méthode via cUrl :

$ curl --header "name:Mickael BARON" http://localhost:9991/api/hello/ENSMA

Bonjour ENSMA de la part de Mickael BARON

Pour retourner l’entête de la réponse , on utilise la requête **GET** suivante :

@GET

@Path("withheaders/{id}")

**public** Response getHelloWithHeaders(@PathParam("id") String id,

@DefaultValue("votre serviteur") @HeaderParam("name") String name) {

**return** Response.ok().header("name", name).entity("Bonjour " + id + " de la part de (voir l'en-tête).").build();

}

En exécutant la commande cURL suivante, on obtient :

curl --header "name:Mickael BARON" http://localhost:9991/api/hello/withheaders/ENSMA -v

\* Trying ::1...

\* TCP\_NODELAY set

\* Connection failed

\* connect to ::1 port 9991 failed: Connection refused

\* Trying 127.0.0.1...

\* TCP\_NODELAY set

\* Connected to localhost (127.0.0.1) port 9991 (#0)

> GET /api/hello/withheaders/ENSMA HTTP/1.1

> Host: localhost:9991

> User-Agent: curl/7.54.0

> Accept: \*/\*

> name:Mickael BARON

>

< HTTP/1.1 200 OK

< name: Mickael BARON

< Content-Type: text/plain

< Content-Length: 46

<

\* Connection #0 to host localhost left intact

Bonjour ENSMA de la part de (voir l'en-tête).

Utiliser @XmlRootElement(name = "train") pour permettre la conversion au format JAVA ⬄ XML de la réponse du webservice.

Généralement, le DAO est contenu dans le cache de la jvm pour les échanges webservices

@XmlRootElement(name = "train")

**public** class Train {

...

}

A voir :

* @XmlAccessorType(XmlAccessType.FIELD)
* @XmlType(name = "DeclarationReportType", propOrder = {

"administration",

"report",

"notes"

})

Contrat de description d’un Webservice REST au format **WADL**

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<application xmlns="http://wadl.dev.java.net/2009/02">

<doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.27 2018-04-10 07:34:57"/>

<doc xmlns:jersey="http://jersey.java.net/" jersey:hint="This is simplified WADL with user and core resources only. To get full WADL with extended resources use the query parameter detail. Link: http://localhost:9991/api/application.wadl?detail=true"/>

<grammars/>

<resources base="http://localhost:9991/api/">

<resource path="hello">

<method id="getHello" name="GET">

<response>

<representation mediaType="text/plain"/>

</response>

</method>

<resource path="{id}">

<param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id" style="template" type="xs:string"/>

<method id="getHello" name="GET">

<request>

<param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="name" style="header" type="xs:string" default="votre serviteur"/>

</request>

<response>

<representation mediaType="text/plain"/>

</response>

</method>

</resource>

<resource path="withheaders/{id}">

<param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id" style="template" type="xs:string"/>

<method id="getHelloWithHeaders" name="GET">

<request>

<param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="name" style="header" type="xs:string" default="votre serviteur"/>

</request>

<response>

<representation mediaType="text/plain"/>

</response>

</method>

</resource>

</resource>

</resources>

</application>

Retourner une requête de type **GET** au format XML ou au format **JSON**

Au format XML : curl http://localhost:9992/api/trains -v

Au format JSON : curl --header "Accept:application/json" http://localhost:9992/api/trains -v

Pour récupérer la réponse au format JSON, ajouter @JsonProperty("departure\_time")

Dans la classe Train contenant l’attribut @XMLRootElement

@XmlRootElement(name = "train")

**public** class Train {

**private** String id;

**private** String departure;

**private** String arrival;

@JsonProperty("departure\_time")

**private** int departureTime; // Format : 1230 = 12h30

...

}

Résumé des requêtes cURL jouées pour requêtes de type GET et POST

# Récupérer la liste des trains.

$ curl --header "Accept: application/json" http://localhost:9992/api/trains

[{"id":"TR123","departure":"Poitiers","arrival":"Paris","departure\_time":1250},{"id":"AX127","departure":"Poitiers","arrival":"Paris","departure\_time":1420},{"id":"PT911","departure":"Poitiers","arrival":"Paris","departure\_time":1710}]

# Créer une réservation de billets de train.

$ curl --header "Accept: application/json" --header "Content-Type: application/json" --request POST --data '{"current\_train":"TR123","number\_places":2}' http://localhost:9992/api/trains/bookings

{"id":"1541683057395","current\_train":"TR123","number\_places":2}

# Récupérer la liste des réservation de billets de train.

$ curl --header "Accept: application/json" http://localhost:9992/api/trains/bookings

[{"id":"1541683057395","current\_train":"TR123","number\_places":2}]

# Récupérer une réservation de billets de train par un identifiant.

$ curl --header "Accept: application/json" http://localhost:9992/api/trains/bookings/1541683057395

[{"id":"1541683057395","current\_train":"TR123","number\_places":2}]

# Supprimer une réservation de billets de train par un identifiant (réussie).

$ curl --request DELETE http://localhost:9992/api/trains/bookings/1541683057395 -v

### Docker/Kubernates

* + **Docker** : logiciel de création de conteneurs pour le déploiement de package EAR/WAR, de micro service, d’environnement, de base de données
  + **Kubernetes** : Logiciel d’orchestration de déploiement de micro services dans des conteneurs
  + **Mini kube** : Outils qui permet de faciliter l’exécution d’un cluster Kuernetes

**Pods** : Plus petite unité de déploiement dans Kubernetes et regroupe plusieurs conteneurs. Permet d’associer un conteneur de type micro service avec plusieurs autres conteneurs utilisés pour le logging, monitoring et communication

**Services :** Propose le load balancing, nommage et l’isolation d’un micro service avec un autre. Les services sont sauvegardées par les controllers de replication, connectés aux Pods

**Labels** : chainage Pods-Services : Nommage et la configuration

---

apiVersion: v1

kind: Service

metadata:

name: shopfront

labels:

app: shopfront

spec:

type: NodePort

selector:

app: shopfront

ports:

- protocol: TCP

port: 8010

name: http

---

apiVersion: v1

kind: ReplicationController

metadata:

name: shopfront

spec:

replicas: 1

template:

metadata:

labels:

app: shopfront

spec:

containers:

- name: shopfront

image: danielbryantuk/djshopfront:latest

ports:

- containerPort: 8010

livenessProbe:

httpGet:

path: /health

port: 8010

initialDelaySeconds: 30

timeoutSeconds: 1

**API Guice**

Framework inventé par Google pour la gestion des microservices

J’ai créé en entrée des controllers REST qui restituaient des fichiers JSON en utilisant les injections de dependances CDI. Les Webservices REST sont déployés depuis un serveur d’application de type Wildfly ou JBOSS.

J’ai injecté des services à partir de points d’injection personnalisés (getter,setter,des champs) et avec des qualificateurs personnalisés (annotation @Produces)

**Annotations**

@Singleton : EJB permettant de définir une instance unique d’une classe JAVA

Similaire à @ApplicationScoped en JAVA SE

Le design pattern Singleton est associé à CDI

Sous spring :

1. **Déclaration d’une resource**

**@Resource**

    @Resource(name="namedFile")

    private File defaultFile;

1. **Création d’un bean**

**@bean**

@Configuration

public class ApplicationContextTestResourceNameType {

    @Bean(name="namedFile")

    public File namedFile() {

        File namedFile = new File("namedFile.txt");

        return namedFile;

    }

}

1. **Création d’un qualifier**

**@Qualifier**

Permet d’injecter des dépendances dans des beans multiples en faisant la correspondance avec le nom de l’attribut qualifier

@Resource

@Qualifier("namedFile")

private File dependency2;

**Creation d’un controller**

@Controller

**public** **class** HomeController {

@Autowired

**private** ProductService productService;

@RequestMapping("/")

**public** String index(Model model) {

model.addAttribute("products", productService.getProducts());

**return** "index";

}

}

**Creation d’une méthode de lancement du webservice**

@SpringBootApplication

@EnableHystrix

**public** **class** ShopfrontApplication {

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

SpringApplication.*run*(ShopfrontApplication.**class**, args);

}

@Bean(name = "stdRestTemplate")

**public** RestTemplate getRestTemplate() {

**return** **new** RestTemplate();

}

}

**Creation dun bean java**

**public** **class** Product {

**private** String id;

**private** String sku;

**private** String name;

**private** String description;

**private** BigDecimal price;

**private** **int** amountAvailable;

**public** Product(String id, String name, String description, BigDecimal price) {

**this**.id = id;

**this**.name = name;

**this**.description = description;

**this**.price = price;

}

**public** Product(String id, String sku, String name, String description, BigDecimal price, **int** amountAvailable) {

**this**.id = id;

**this**.sku = sku;

**this**.name = name;

**this**.description = description;

**this**.price = price;

**this**.amountAvailable = amountAvailable;

}

**public** String getId() {

**return** id;

}

}

**Creation d’un sous controller**

@RestController

@RequestMapping("/products")

**public** **class** ProductResource {

@Autowired

**private** ProductService productService;

@RequestMapping()

**public** List<Product> getProducts() {

**return** productService.getProducts();

}

}

**Creation dun service**

@Service

**public** **class** ProductService {

@Autowired

**private** StockRepo stockRepo;

@Autowired

**private** ProductRepo productRepo;

**public** List<Product> getProducts() {

Map<String, ProductDTO> productDTOs = productRepo.getProductDTOs();

Map<String, StockDTO> stockDTOMap = stockRepo.getStockDTOs();

// Merge productDTOs and stockDTOs to a List of Products

**return** productDTOs.values().stream()

.map(productDTO -> {

StockDTO stockDTO = stockDTOMap.get(productDTO.getId());

**if** (stockDTO == **null**) {

stockDTO = StockDTO.***DEFAULT\_STOCK\_DTO***;

}

**return** **new** Product(productDTO.getId(), stockDTO.getSku(), productDTO.getName(), productDTO.getDescription(), productDTO.getPrice(), stockDTO.getAmountAvailable());

})

.collect(Collectors.*toList*());

}

**public** List<Product> productsNotFound() {

**return** Collections.***EMPTY\_LIST***;

}

}

**Creation dun fichier yaml**

---

apiVersion: v1

kind: Service

metadata:

name: productcatalogue

labels:

app: productcatalogue

spec:

type: NodePort

selector:

app: productcatalogue

ports:

- protocol: TCP

port: 8020

name: http

---

apiVersion: v1

kind: ReplicationController

metadata:

name: productcatalogue

spec:

replicas: 1

template:

metadata:

labels:

app: productcatalogue

spec:

containers:

- name: productcatalogue

image: danielbryantuk/djproductcatalogue:1.0

ports:

- containerPort: 8020

livenessProbe:

httpGet:

path: /healthcheck

port: 8025

initialDelaySeconds: 30

timeoutSeconds: 1

**Un bean** , un service, un composant : classes java

Un service est toujours relie a une interface

Si il y a plusieurs implementations de l’interface alors on doit utiliser l’annotation @Qualifier pour dire quelle implementation on doit injecter.

Bean : composant spring, composant injectable

**@Qualifier** : permet de lever l’ambiguité lors d’injection de bean

**@Autowired** : a utiliser sans @qualifier si pas d’ambiguité

**@Bean :** déclaré sur une méthode pour injecter une classe injectable

**@Component** : Composant générique pour tout type de spring managed composant

**@Repository** : classe qui remplit le rôle de Data Access Object ou DAO (persistence layer)

**@Service** : spécialisation de la balise @Component, classe java qui produit des fonctionnalités business (service layer)

**@Controller :** Composant utilisé dans la couche de présentation (presentation layer)

**@RequestMapping** utilisé en décoration de la classe Composant avec le chemin d’accès via un appel REST). Si utilisé

Lors de l’injection d’un service, une instance est créé, elle peut être créé sous 4 Scopes différents :

* **@Singleton :** retourne une seul type d’instance par Spring IoC Container
* **@Request**: Returns a single instance for every HTTP request call.

@Bean

@RequestScope

public HelloMessageGenerator requestScopedBean() {

    return new HelloMessageGenerator();

}

* **@Session** Returns a single instance for every HTTP session

@Bean

@SessionScope

public HelloMessageGenerator sessionScopedBean() {

    return new HelloMessageGenerator();

}

* **@Prototype :** retourne une instance différente à chaque fois que requêté depuis le container

@Bean

@Scope("prototype")

public Person personPrototype() {

    return new Person();

}

### Template de création de micro services :

### Avec Spring Boot

* **Lancement du micro service**

L’annotation **@SpringBootApplication** encapsule le lancement du serveur Tomcat et sa configuration

@SpringBootApplication

@EnableHystrix

**public** **class** ShopfrontApplication {

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

SpringApplication.*run*(ShopfrontApplication.**class**, args);

}

@Bean(name = "stdRestTemplate")

**public** RestTemplate getRestTemplate() {

**return** **new** RestTemplate();

}

}

* **Controller**

@Controller

**public** **class** HomeController {

@Autowired

**private** ProductService productService;

@RequestMapping("/")

**public** String index(Model model) {

model.addAttribute("products", productService.getProducts());

**return** "index";

}

}

* **Beans**
* **Repository**

@Component

**public** **class** ProductRepo {

@Value("${productCatalogueUri}")

**private** String productCatalogueUri;

@Autowired

@Qualifier(value = "stdRestTemplate")

**private** RestTemplate restTemplate;

**public** Map<String, ProductDTO> getProductDTOs() {

ResponseEntity<List<ProductDTO>> productCatalogueResponse =

restTemplate.exchange(productCatalogueUri + "/products",

HttpMethod.***GET***, **null**, **new** ParameterizedTypeReference<List<ProductDTO>>() {

});

List<ProductDTO> productDTOs = productCatalogueResponse.getBody();

**return** productDTOs.stream()

.collect(Collectors.*toMap*(ProductDTO::getId, Function.*identity*()));

}

}

* **Resource**

@RestController

@RequestMapping("/products")

**public** **class** ProductResource {

@Autowired

**private** ProductService productService;

@RequestMapping()

**public** List<Product> getProducts() {

**return** productService.getProducts();

}

* **Service**

@Service

**public** **class** ProductService {

@Autowired

**private** StockRepo stockRepo;

@Autowired

**private** ProductRepo productRepo;

**public** List<Product> getProducts() {

Map<String, ProductDTO> productDTOs = productRepo.getProductDTOs();

Map<String, StockDTO> stockDTOMap = stockRepo.getStockDTOs();

// Merge productDTOs and stockDTOs to a List of Products

**return** productDTOs.values().stream()

.map(productDTO -> {

StockDTO stockDTO = stockDTOMap.get(productDTO.getId());

**if** (stockDTO == **null**) {

stockDTO = StockDTO.***DEFAULT\_STOCK\_DTO***;

}

**return** **new** Product(productDTO.getId(), stockDTO.getSku(), productDTO.getName(), productDTO.getDescription(), productDTO.getPrice(), stockDTO.getAmountAvailable());

})

.collect(Collectors.*toList*());

}

**public** List<Product> productsNotFound() {

**return** Collections.***EMPTY\_LIST***;

}

}

### Avec Spring sans SpringBoot

* **Lancement du micro service**

On définit manuellement la méthode **run**

Elle se base sur la création d’un **Environment,** on utilise l’API **Guice** de Google pour la création d’un injecteur

**public** **class** ProductServiceApplication **extends** Application<ProductServiceConfiguration> {

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

**new** ProductServiceApplication().run(args);

}

@Override

**public** String getName() {

**return** "product-list-service";

}

@Override

**public** **void** initialize(Bootstrap<ProductServiceConfiguration> bootstrap) {

// nothing to do yet

}

@Override

**public** **void** run(ProductServiceConfiguration config,

Environment environment) {

**final** BasicHealthCheck healthCheck = **new** BasicHealthCheck(config.getVersion());

environment.healthChecks().register("template", healthCheck);

Injector injector = Guice.*createInjector*();

environment.jersey().register(injector.getInstance(ProductResource.**class**));

}

}

* **Beans**
* **HealthCheck**

**public** **class** BasicHealthCheck **extends** HealthCheck {

**private** **final** String version;

**public** BasicHealthCheck(String version) {

**this**.version = version;

}

@Override

**protected** Result check() **throws** Exception {

**return** Result.*healthy*("Ok with version: " + version);

}

}

* **Resource**

@Path("/products")

@Produces(MediaType.***APPLICATION\_JSON***)

**public** **class** ProductResource {

**private** ProductService productService;

@Inject

**public** ProductResource(ProductService productService) {

**this**.productService = productService;

}

@GET

@Timed

**public** Response getAllProducts() {

**return** Response.*status*(200)

.entity(productService.getAllProducts())

.build();

}

@GET

@Timed

@Path("{id}")

**public** Response getProduct(@PathParam("id") String id) {

Optional<Product> result = productService.getProduct(id);

**if** (result.isPresent()) {

**return** Response.*status*(Response.Status.***OK***)

.entity(result.get())

.build();

} **else** {

**return** Response.*status*(Response.Status.***NOT\_FOUND***)

.build();

}

}

}

* **Service**

**public** **class** ProductService {

**private** Map<String, Product> fakeProductDAO = **new** HashMap<>();

**public** ProductService() {

fakeProductDAO.put("1", **new** Product("1", "Widget", "Premium ACME Widgets", **new** BigDecimal(1.20)));

**public** List<Product> getAllProducts() {

**return** **new** ArrayList<>(fakeProductDAO.values());

}

**public** Optional<Product> getProduct(String id) {

**return** Optional.*ofNullable*(fakeProductDAO.get(id));

}

}

# Cours JAVA

### **Collections**

* List : collection d'éléments ordonnés qui accepte les doublons
* Set : collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons
* Map : collection sous la forme d'une association de paires clé/valeur
* Queue et Deque : collections qui stockent des éléments dans un certain ordre avant qu'ils ne soient extraits pour traitement

Les fonctionnalités des collections sont définies dans cinq interfaces de base : Collection, List, Set, Map, Queue.

|  |  |  |
| --- | --- | --- |
| Set collection d'éléments uniques | List collection avec doublons | Map collection sous la forme clé/valeur |
| Tableau redimensionnable |  | ArrayList, Vector (JDK 1.1) |  |
| Arbre | TreeSet |  | TreeMap |
| Liste chaînée |  | LinkedList |  |
| Collection utilisant une table de hachage | HashSet |  | HashMap, Hashtable (JDK 1.1) |
| Classes du JDK 1.1 |  | Stack |  |

### **Functional interface**

Une « Functional interface » possède les propriétés suivantes :

* Peut posséder l’annotation @FunctionalInterface
* Possède une et une seule **one abstract non object méthod**

**public** **interface** Discounter {

BigDecimal applyDiscount(BigDecimal amount);

**boolean** equals(Object obj);

}

Appel de l’interface fonctionnelle :

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

Discounter easterDiscounter = amount -> amount.multiply(BigDecimal.*valueOf*(0.5));

Discounter christmasDiscounter = amount -> amount.multiply(BigDecimal.*valueOf*(0.8));

List<Discounter> list =

Stream

.*of*(easterDiscounter, christmasDiscounter)

.collect(Collectors.*toList*());

## **Non-Static Nested Classes / Inner class**

* Tout type d’accessibilité dans leur déclarations
* Associées à une instance de la classe encapsulante
* Accès à tous les membre de la classe encapsulante, Static ou non static
* Ne définissent que des membres non statiques

Déclaration d’une inner class :

Outer outer = new Outer() ;

Outer.Inner inner = outer.new Inner() ;

### Static Nested Class

* Rattachée à la classe encapsulante, pas à une instance de la classe
* Tout type d’accessibilité dans leur déclaration
* Uniquement accès aux membres statiques de la classe encapsulante
* Définissent des membres statiques et non statiques

Déclaration d’une static nested class :

Outer.Inner innter = new Outer.Innter()

### Différence entre Static Nested Class / Non static Nested class (Inner Class)

* Les Inner class sont rattachés à la classe encapsulante et ont donc accès à leur membres

### Local class

* Type spécial de inner class, définie directement dans le corps d’une méthode
* N’ont pas d’accessibilité définie dans leur déclaration
* Accès aux membres statiques et non statiques dans la classe encapsulante
* Ne définissent que des membres d’instance

A utiliser si besoin de :

* Créer plus d’une instance de class
* Accéder à son constructeur
* Introduire un nouveau named type
* Utilisation Pattern Singleton dans une inner class

<https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>

* Approche ThreadSafe : la classe privée etnested, contient une instance de la classe singleton. Quand le singleton est chargé, il ne l’est pas en mémoire mais que lorsque la méthode getinstance est appelé
* Ne requiert pas de synchronisation (Stack memory ?)

**public** **class** Handler {

**public** String nonStaticVariable = "nonStaticVariable";

**public** **static** String *staticVariable* = "staticVariable";

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

RankNode root = **new** RankNode(20);

//Call inner class (non static nested)

Handler outer = **new** Handler();

Handler.InnerClass inner = outer.**new** InnerClass();

//Call static nested class

StaticNestedClass staticNested = **new** Handler.StaticNestedClass();

**int** rank = root.getRank(23);

}

**public** String treatmentWithLocalClass() {

//Local Class : accès aux membres statiques et non statiques

**class** LocalClass {

Handler handler = **new** Handler();

String localMethod() {

handler.nonStaticVariable = "1";

handler.*staticVariable* = "1";

**return** handler.nonStaticVariable + handler.*staticVariable*;

}

}

LocalClass localClass = **new** LocalClass();

**return** localClass.localMethod();

}

//Pour une inner Class

**public** **class** InnerClass {

//Acces aux membres non statiques uniquement

**public** String getOuterNonStaticVariable() {

**return** Handler.**this**.nonStaticVariable;

}

}

**public** **static** **class** StaticNestedClass {

//Acces aux membres statiques et non statiques

**public** String getOuterVariable() {

Handler handler = **new** Handler();

handler.nonStaticVariable = "1";

handler.*staticVariable* = "1";

**return** Handler.*staticVariable*;

}

}

//Singleton class

**private** Handler() {}

**private** **static** **class** SingletonHelper{

**private** **static** **final** Handler ***instance*** = **new** Handler();

}

**public** **static** Handler getInstance() {

**return** SingletonHelper.***instance***;

}

}

### Static Method

Une méthode statique et une **Utility method** ou **Helper method,** associée à une classe ou interface. Elle n’est **pas associée** à un **objet**

Une classe est déclarée statique dès qu’elle est **chargée en mémoire** en **dehors de toute instanciation.**

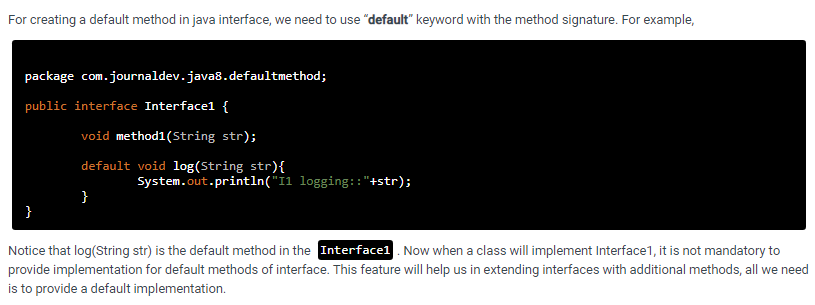
Quelque soit le nombre d’instanciation de la classe, un attribut de classe n’existe en un seul exemplaire.

Besoin :

* Garder les Helper et Utility méthodes spécifiques à une interface dans la même interface plutôt que dans une classe de type Utility
* Claire séparation des responsabilités. Pas besoin d’avoir une classe Utility pour garder toutes les méthodes Utility de l’API Collection
* Facile d’etendre les API
* Facile de maintenenir les API

### Default Method

* Avec Java 8, les **méthodes Default** permettent d’être déclarées dans les interfaces sans affecter les classes qui implémentent les interfaces. On peut donc ajouter des méthodes sans briser toute la structure des classes d’implémentation.
* Avant java 8, les interfaces ne pouvaient qu’avoir des méthodes abstraites.



Pas besoin de les override dans les classes d’implémentations.

Si une classe implémente **une seule interface :**

//Appel de la méthode log déclarée par l'interface 1, lorsque la clase Main n'implémente qu'une seule interface

**public** **void** handler() {

log("appel de la méthode log depuis interface ?");

}

* Si une classe implémente **plusieurs interfaces : Diamond problem :** 
  + Override la méthode déclarée dans les deux interfaces, dans la classe d’implémentation

//Si plusieurs interfaces, diamond problem, on doit implémenter la method log dans la classe d'implementation

**public** **void** log() {

Interface1.**super**.log("appel de la méthode log depuis interface 1");

}

* Finalement, dans une inner class, on pourra appeler la méthode log depuis l’interface :

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

Main main = **new** Main();

Main.Inner inner = main.**new** Inner();

inner.log("chaine de caractère");

}

//Création d'une classe Inner implémentant les deux interfaces

**public** **class** Inner **implements** Interface1, Interface2{

//Appel de la méthode log déclarée par l'interface 1, lorsque la clase Main n'implémente qu'une seule interface

**public** **void** handler() {

Main.**this**.log("appel de la méthode log depuis interface ?");

}

@Override

**public** **void** method1() {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** method2() {

// **TODO** Auto-generated method stub

}

//Méthode appelée depuis la méthode Main lorsque la inner class implémente plusieurs interfaces

//Il est nécessaire d'override la méthode log.

@Override

**public** **void** log(String str) {

Interface2.**super**.log(str);

}

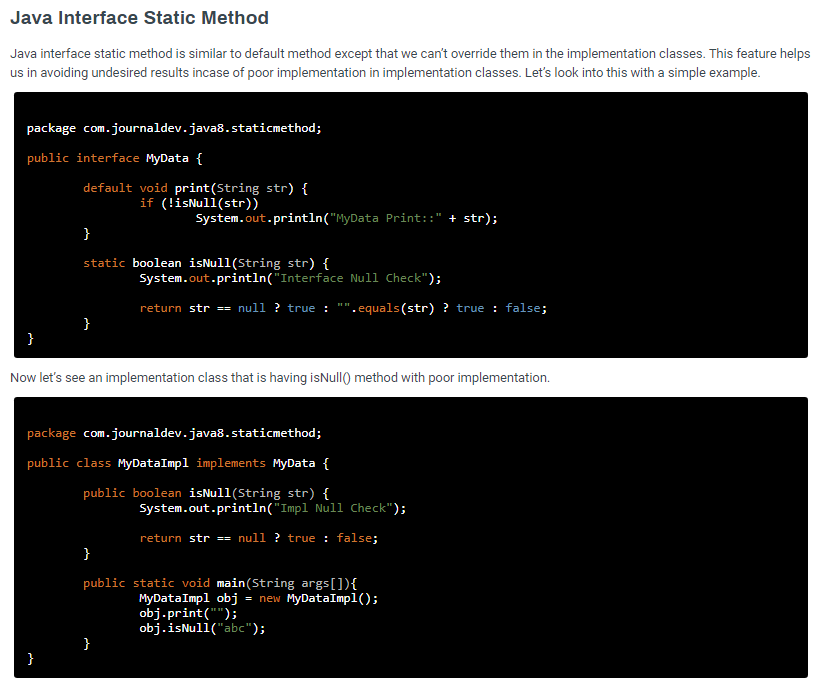
}

### Static final / Static

La différence entre une final et une static final, c'est que le static final ne sera qu'une fois en mémoire (donc c'est mieux si elle doit avoir la même valeur pour toutes les instances de ta classe).

### Static Method

* On ne peut pas les override dans les classes d’implémentations, elles ne supportent pas l’héritage de classe.
* Les methods statiques d’interfaces java sont visible dans les methods d’interface uniquement.
* Les méthodes statiques d’interface JAVA sont similaires aux Java interface default méthod sauf qu’on ne pas les override dans les classes d’implémentations.



Static methodes sont utiles produire des **méthodes utilitaires**

Static méthodes permet d’ajouter de la sécurité car non accessibles à partir d’une interface implémentée par la classe.

### **Autoboxing / Unboxing**

**Autoboxing:**

**Converti une valeur primitive dans un objet de la classe Wrapper correspondante.**

Exemple : Convertit **int** en **integer Class.**

La JVM utilise **l’autoboxing** quand un type primitif est:

* Passé en paramètre d’une méthode **qui attend un objet de sa classe Wrapper correspondante.**
* Assigné à une variable de la Wrapper class correspondante.

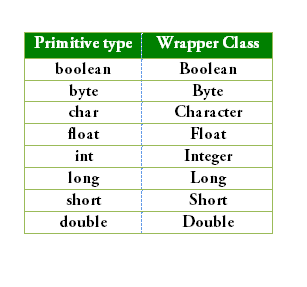
**Unboxing**

Converti **un objet de sa classe Wrapper** dans un **type** **primitif.**

Exemple : Converti Integer class en int

La JVM utilise **l’unboxing** quand un objet de sa classe Wrapper est :

* Passé en paramètre d’une méthode qui attend un type primitif.
* Assigné à une variable dans le type primitif corresponda

[](https://media.geeksforgeeks.org/wp-content/uploads/Wrapper.png)

**Avantages de l’autoboxing / unboxing**

Permet d’écrire du code plus propre et plus facile à lire

La technique permet de rendre interchangeable l’utilisation de type primitifs et des objets de classes Wrapper, **pas de Cast explicite**

Outils :

                public static int sumOfOddNumber(List<Integer> list)

    {

        int sum = 0;

        for (Integer i : list)

        {

            // unboxing of i automatically

            if(i % 2 != 0)

                sum += i;

            /\* unboxing of i is done automatically

               using intvalue implicitly

            if(i.intValue() % 2 != 0)

                sum += i.intValue();\*/

        } return sum; }

### Différences entre JVM JRE JDK

<https://www.baeldung.com/jvm-vs-jre-vs-jdk>

JVM : Implémentation de la machine virtuelle qui compile du code java sous forme de bytecode.

* + - Stocke l’information des classes dans la mémoire.
    - Exécute le bytecode généré par le compilateur java

Composants d’une JVM :

* + - Classe Loaders
    - Run-time Data Areas
    - Exécution engine

Définit différents types de mémoire :

* + - Method area / Permanent Generation Space : stocke les structures comme run-time constant pool, fields et method data, le code pour les méthodes et les constructeurs

Créé quand la JVM démarre

### **Garbage Collector**

Java takes care of memory management using Garbage Collection. It's a process of looking at heap memory, identifying which objects are in use and which are not, and finally deleting unused objects.

GC is a daemon thread. It can be called explicitly using System.gc() method, however, it won't be executed immediately and the JVM decides when to invoke GC.

JRE : **Java Runtime Environment (JRE) is a bundle of software components used to run Java applications.**

Core components of the JRE include:

* An implementation of a Java Virtual Machine (JVM)
* Classes required to run the Java programs
* Property Files

We discussed the JVM in the above section. Here we will focus on the core classes and support files.

### Types d’espaces de stockage

Trois modes d’allocation de mémoire lors de la vie d’un programme :

1. allocation statique (segment de donnée) : variables allouées pour toute la durée de vie du programme (variable static ou globale) est stocké dans le segment de donnée. Leur présence est prévue au moment de la compilation du programme : un emplacement mémoire leur est réservée dès le lancement du programme

2. allocation automatique, sur la pile (« stack allocation ») : variables locales et arguments effectifs d’une fonction est stockée dans la pile, crées « automatiquement » lors de l'exécution de la déclaration, et détruit « automatiquement » à la fin du bloc.

3. allocation dynamique, sur le tas (« heap allocation ») : variable allouée à l’exécution du programme (instruction new en java, malloc en C), est stockée dans le tas (heap). L'OS gère la taille de la pile et du tas automatiquement, selon : ¬ les besoins du programme, ¬ les ressources disponibles

### Java interview questions

### <https://www.javacodegeeks.com/java-interview-questions.html>

### Thread

Un ***thread*** désigne un « **fil d'exécution** » dans le programme ; c'est-à-dire une suite linéaire et continue d'instructions qui sont exécutées séquentiellement les unes après les autres. En fait, le langage Java est *multi-thread*, c'est-à-dire qu'il peut faire cohabiter plusieurs fils d’exécution de façon indépendante.

### Mémoire globale

<https://www.codeflow.site/fr/article/java-stack-heap>

La mémoire sélectionne deux types de mémoire différentes pour des opérations du type :

* Déclaration de nouvelles variables et d’objets
* Appel de nouvelles methods
* Déclarer un String

### Stack Memory

La mémoire de pile en Java est utilisée pour l’allocation de mémoire statique et l’exécution d’un thread. \*\* Elle contient des valeurs primitives spécifiques à une méthode et des références à des objets contenus dans un segment de mémoire, référencées à partir de la méthode.

Utilisé pour l’allocation de mémoire statique et de l’exécution de thread.

Se dimensionne en fonction des appels de methodes effectués

* Les variables dans la stack n’existent que lorsque la méthode est en cours d’exécution
* Alloué et désalloué automatiquement quand la méthode finit son exécution
* Accès rapide comparé à la Heap Memory et retourne une erreur si full
* Mémoire threadSafe puisque chaque thread opère dans sa stack

### Heap Memory

Utilisé pour l’allocation de mémoire dynamique, pour le chargement des JRE et des objets java lors de leur exécution. Les nouveaux objets sont créés dans le Heap Space mais les références sont stockées dans la Stack Memory.

* It's accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation
* Plus lent d’accéder à ce type de mémoire que la stack Memory
* N’est pas désalloué automatiquement : Le Garbage Collector a besoin de libérer de l’espace sur les objets non utilisés pour garder de l’efficacité
* Contrairement à la Stack Memory, n’est pas thread-safe et a besoin d’effectuer des synchronisations.

### Abstract class / Interfaces

Comme pour les interfaces, **une classe abstraite ne peut pas être instanciée.**

Animal unAnimal; // OK

unAnimal = new Animal (...); // ERREUR

Contrairement aux interfaces, une classe abstraite peut contenir des méthodes concrètes.

Contrairement aux interfaces, une classe abstraite peut ne pas contenir de méthodes abstraites

Projet Mga

**public** **abstract** **class** AbstractStatistiqueGeneralTotalVO **extends** AbstractStatistiqueGeneralVO {

/\*\*

\* Date de d?but de la p?riode s?lectionn?e

\*/

**private** String dateDebut;

/\*\*

\* Date de fin de la p?riode s?lectionn?e

\*/

**private** String dateFin;

/\*\*

\* Libell? de l'unit? mon?taire s?lectionn? par l'utilisateur

\*/

**private** String uniteMonetaire;

**protected** AbstractStatistiqueGeneralParUAVO currentUA;

/\*\*

\* Liste des totaux pour chacune des UA s?lectionn?es par l'utilisateur

\*/

**protected** List listeStatsUA = **new** ArrayList();

/\*\*

\* **@return** Renvoie dateDebut.

\*/

**public** String getDateDebut() {

**return** dateDebut;

}

/\*\*

\* **@param** dateDebut dateDebut ? d?finir.

\*/

**public** **void** setDateDebut(String dateDebut) {

**this**.dateDebut = dateDebut;

}

}

Appel de la méthode setDateFin via une classe étendant la classe abstraite

**public** **class** StatistiqueGeneralTotalParLigneVO **extends** AbstractStatistiqueGeneralTotalVO {

/\*\*

\* **@param** dateDebut

\* **@param** dateFin

\* **@param** uniteMonetaire

\* **@throws** ParametreInvalideException

\* **@throws** ParametreNulException

\*/

**public** StatistiqueGeneralTotalParLigneVO (){

}

**public** StatistiqueGeneralTotalParLigneVO(String dateDebut, String dateFin, String uniteMonetaire)

**throws** ParametreInvalideException, ParametreNulException {

setDateDebut(dateDebut);

setDateFin(dateFin);

setUniteMonetaire(uniteMonetaire);

}

## Programmation concurrent

### Bloc synchronized

Quand un thread obtient cette clé, il entre dans le bloc, et exécute le code qui s'y trouve. À la sortie du bloc, il rend la clé à l'objet qui garde le bloc.

Aucun thread ne peut entrer dans ce bloc s'il ne possède pas la clé qui permet d'y entrer.

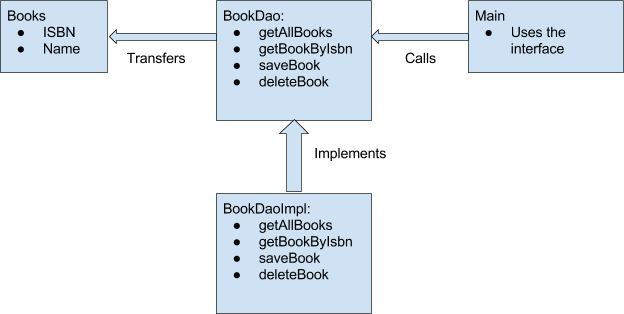
Supposons qu'un thread se présente à l'entrée du bloc, alors qu'un premier est dedans, en train de l'exécuter. La clé n'étant pas disponible, cet autre thread se verra refuser l'accès au bloc. Il devra attendre que le premier thread rende la clé afin de pouvoir y entrer à son tour.

## Design Patterns

Feuille de route : <https://www.javatpoint.com/design-patterns-in-java>

### Dao Design Patterns

<https://www.journaldev.com/16813/dao-design-pattern>



Fonctionnalités :

* Si changement d’un mécanisme de persistence :
  + La couche de service n’a pas besoin de savoir l’origine de la données.
  + Si un changement est opéré sur le socle technique de la BDD, la seule modification à effectuer est dans la couche DAO.
* Faible couplage entre les composants de l’applications :
  + Couche de Vue n’a aucune dépendance avec la Couche DAO
  + Seule la couche de Service a une dépendance avec la couche DAO, et seulement avec les interfaces, pas d’implémentation concrète.

### Factory Design Pattern Projet MGA

### DAO Design Pattern Projet MGA

Projet Mga

* **Couche** Facade

**public** **class** PrpDestructionColisTrocadeFacade {

* **Couche** VO
  + Interface VO

**public** **interface** PrpDestructionColisTrocadeAction {

* + Classe java VO implémentant l’interface VO

**public** **class** PrpDestructionColisTrocadeActionImpl **implements** PrpDestructionColisTrocadeAction{

…

**public** DestructionColisTrocadeVO performReal(

DestructionColisTrocadeVO destructionColisTrocadeVO, User user, **int** statut) {

* **Couche** BS
  + Interface BS

**public** **interface** IDestructionColisTrocadeBS {

**public** List<ColisTrocadeBO> trouverColisParAtelier(String atelier)

**throws** AccesBaseException, AttributException, PersistenceException;

**public** **void** majColisTrocade(Date date, String typeColis, String sscc)

**throws** PersistenceException, AccesBaseException, IdentifiantGS1Exception, AttributException, EnregistrementInconnuException;

* + Classe java BS implémentant l’interface BS

**public** **class** DestructionColisTrocadeBS **extends** GCMBS<ColisTrocadeBO> **implements** IDestructionColisTrocadeBS {

* **Couche** DS
  + Interface DS

**public** **interface** IColisClientDS {

* + Classe Java DS implémentant l’interface DS

**public** **class** ColisClientDS

**implements** IColisClientDS {

**public** ColisClientBO trouverColis(String idSSCC, ITransaction tnx)

**throws** AccesBaseException, AttributException, IdentifiantGS1Exception {

IdentifiantGS1Util.*verifierIdSSCCGS1*(idSSCC);

**return** getColisDAO().trouverColis(idSSCC, tnx);

}

* **Couche** DAO
  + Interface DAO

**public interface IColisDAO {**

/\*\* récupération du colis à partir d'un identifiant SSCC \*/

**public** ColisClientBO trouverColis(String idSSCC, ITransaction tnx)

**throws** AccesBaseException, AttributException, IdentifiantGS1Exception;

* + Classe java DAO

**public** **class** ColisClientDS

**implements** IColisClientDS {

**public** ColisClientBO trouverColis(String idSSCC, ITransaction tnx)

**throws** AccesBaseException, AttributException, IdentifiantGS1Exception {

IdentifiantGS1Util.*verifierIdSSCCGS1*(idSSCC);

**return** getColisDAO().trouverColis(idSSCC, tnx);

}

## Creational Pattern

### F

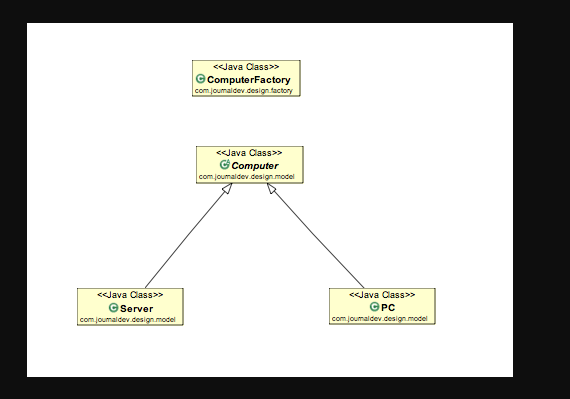
Git : git clone <https://github.com/journaldev/journaldev.git>

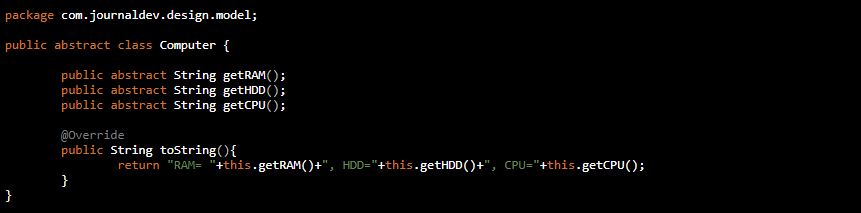
Doc :

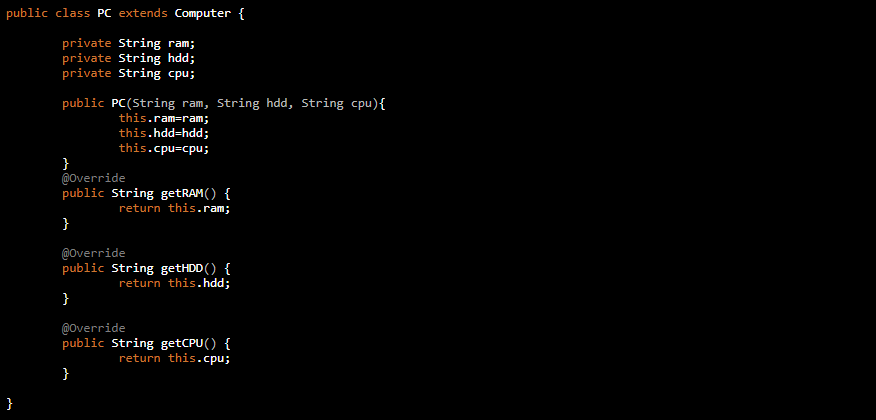
<https://www.journaldev.com/1392/factory-design-pattern-in-java>

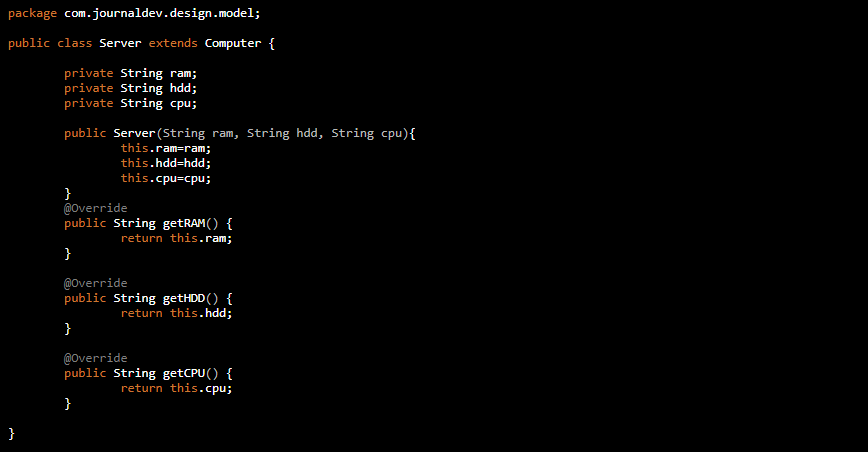
<https://www.javatpoint.com/factory-method-design-pattern>

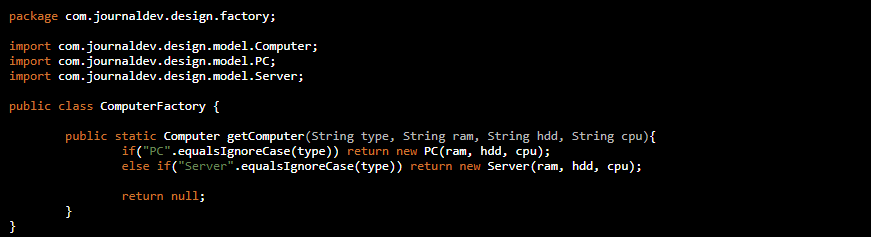
* Une classe **ComputerFactory** indépendante des autres classes
* Une classe **Computer** pouvant être : **Abstraite**, une **interface** ou une classe standard
* Des sous classes **PC** et **Server** étendant la classe principale **Computer**











Utilité :

* Quand une classe ne sait pas quelle sous-classe serait necessaire de créer
* Quand une classe veut que ses sous classes spécifient quell objet créer
* Quand la parent classe choisit la creation d’objet à réaliser par ses sous classes

Avantages :

* Permet de laisser les sous classes décider quel objet créer
* Le design Pattern Factory utilise du code pour les interfaces plutôt que pour l’ implémentation.
* Plus d’instanciation de l’actuelle implémentation des classes depuis le code client, Les classes à utiliser par le code client héritent directement d’une classe abstraite / interface. Le code est donc :
  + Plus robuste
  + Moins Couplé
  + Facilement extensible

*L’objet Calendar utilise le design Pattern Factory*

Projet Mga

Classe principale (**Computer ⬄ LogManagerFactory**), utilisée pour sa propre instanciation (Pattern **Singleton**) :

**public** **class** LogManagerFactory {

\*

\* Acc?s au singleton

\***@return** Une instance unique de la classe

\*/

**public** **static** LogManagerFactory getInstance() {

**if** (mLogManagerFactory == **null**) {

**synchronized** (LogManagerFactory.**class**) {

mLogManagerFactory = **new** LogManagerFactory();

}

}

}

}

### Singleton Design Pattern

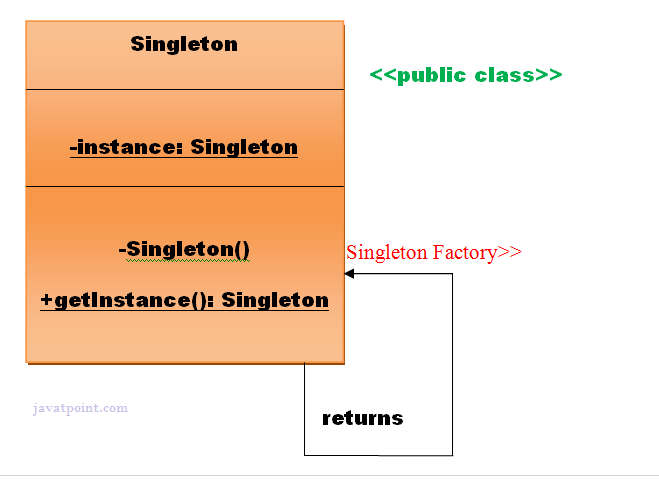
Doc

<https://www.javatpoint.com/singleton-design-pattern-in-java>

<https://sandeepdass003.wordpress.com/2018/02/23/eager-and-lazy-instantiation-in-singleton-design-pattern-implementation/>

<https://sandeepdass003.wordpress.com/2017/09/03/singleton-in-java/>

/ !\ Notion Thread-Safe / !\



Deux formes de singleton design pattern :

* **Eager Instantiation:** creation d’une instance au chargement des classes
* **Lazy Instantiation :** création d’une instance quand requis

Pour la construction du design pattern il faut :

* **Membre statique** : contient l’instance de la classe singleton et chargé une seule fois en mémoire car statique.

Private Static EagerInitializedSingleton = new EagerInitializedSingleton()

* **Private constructor**: Empêche à l’application cliente d’utiliser le constructeur

Private EagerInitializedSingleton(){}

* **Statique Factory Method**: Point d’accès global à l’objet Singleton et retourne une instance à l’appeleur.

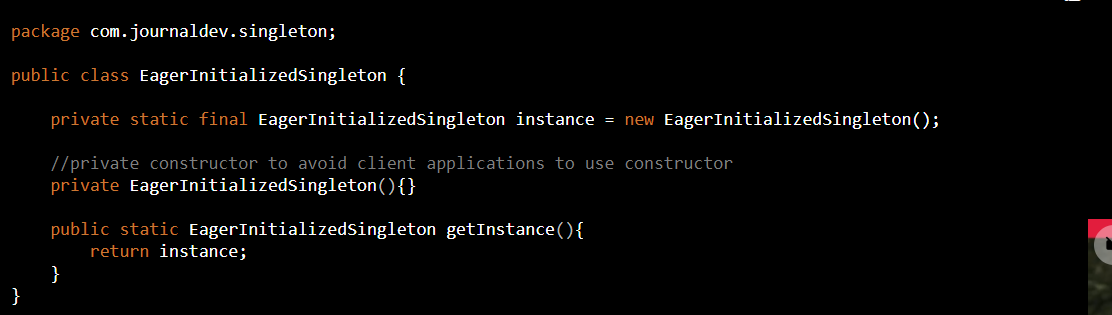
public static EagerInitializedSingleton getInstance(){

return instance;

}

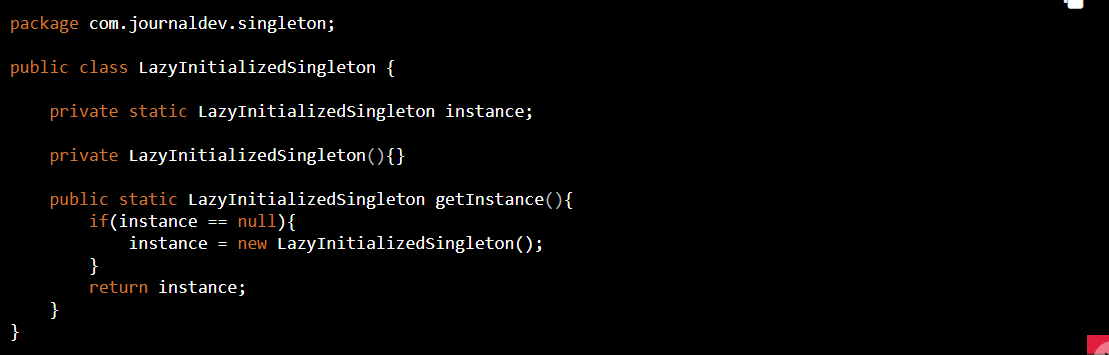
**Eager initialization**

Une instance de la classe Singleton est créée au **chargement de la classe**.

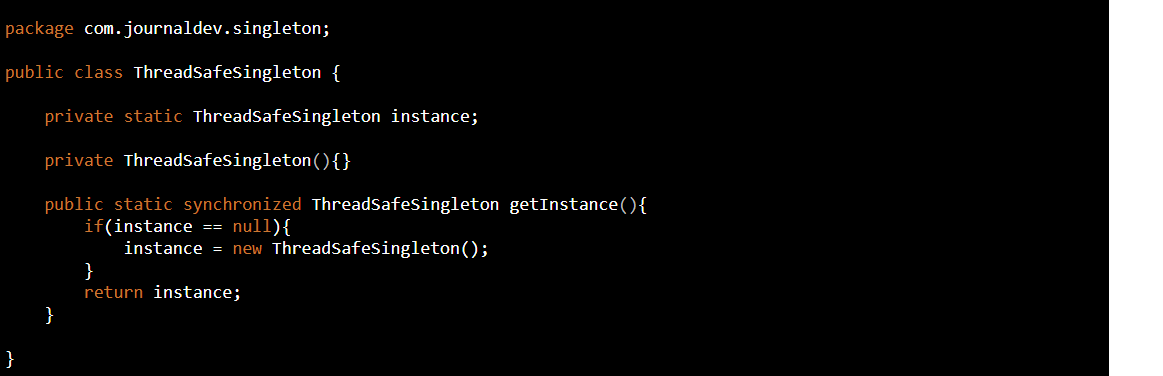


**Lazy initialization**

Une instance de la classe Singleton est créée une seule fois grâce à l’attribut **Synchronized** et au **constructeur privé**



**Thread Safe initialization**



Avantages :

* Sauve de la mémoire car l’objet n’est pas créé à chaque requête. Une seule et même instance est réutilisé encore et encore

Besoin :

* Surtout utilisé pour le multi thread et en application de base de données.
  + Utilisé pour le logging, caching, thread pool …

**Projet MGA**

La méthode getInstance permet d’instancier la classe Factory

Public class LogManagerFactory {

/\*\*

\*/

**private** LogManagerFactory() {

URL config = ClassLoader.*getSystemResource*(***LOG4J\_PROPERTIES\_FILE***);

PropertyConfigurator.*configure*(config);

}

/\*\*

\* Acc?s au singleton

\***@return** Une instance unique de la classe

\*/

**public** **static** LogManagerFactory getInstance() {

**if** (mLogManagerFactory == **null**) {

**synchronized** (LogManagerFactory.**class**) {

mLogManagerFactory = **new** LogManagerFactory();

}

}

**return** mLogManagerFactory;

}

}

**public** **class** Mft {

**…**

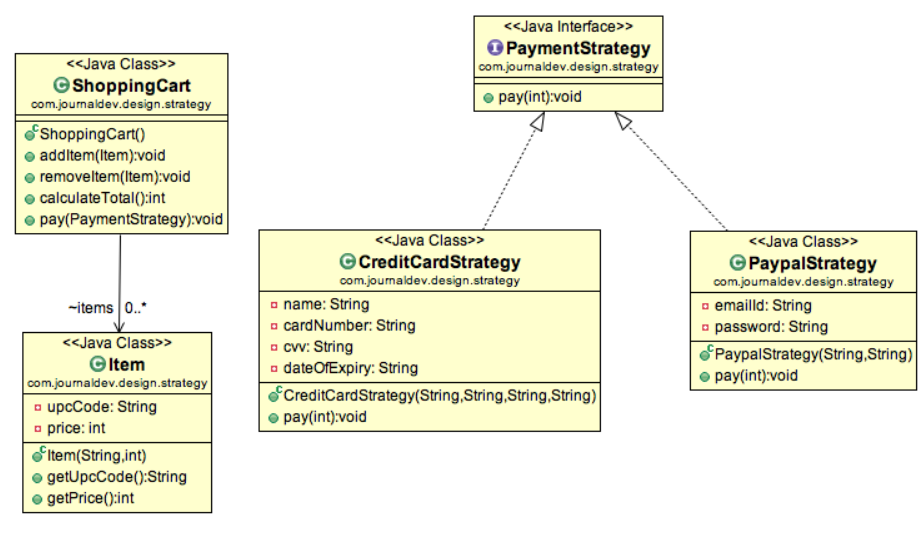
**private** **static** LogManager logger = LogManagerFactory.getInstance().getLogManager(Mft.**class**);

### Strategy Design Pattern

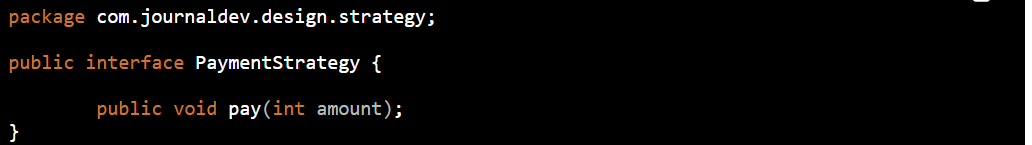
Doc

<https://www.journaldev.com/1754/strategy-design-pattern-in-java-example-tutorial>

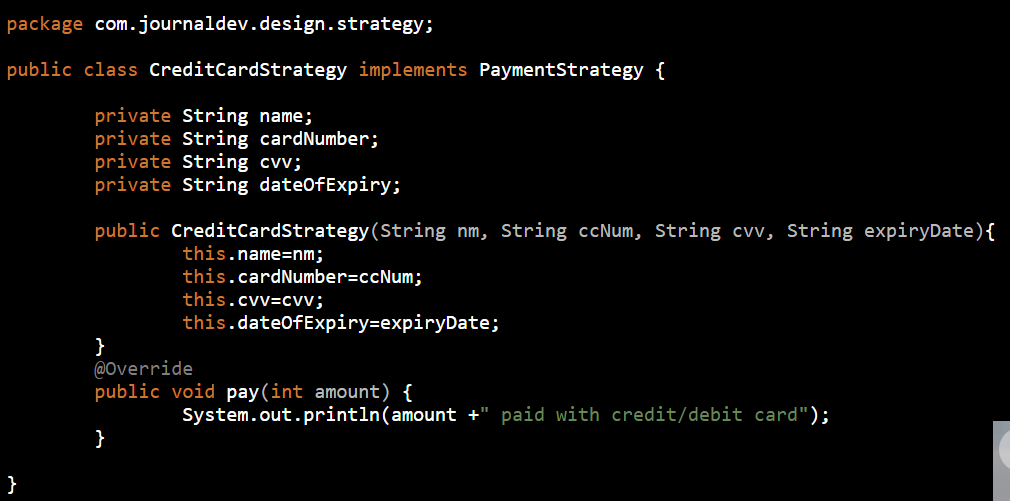
<https://github.com/iluwatar/java-design-patterns>

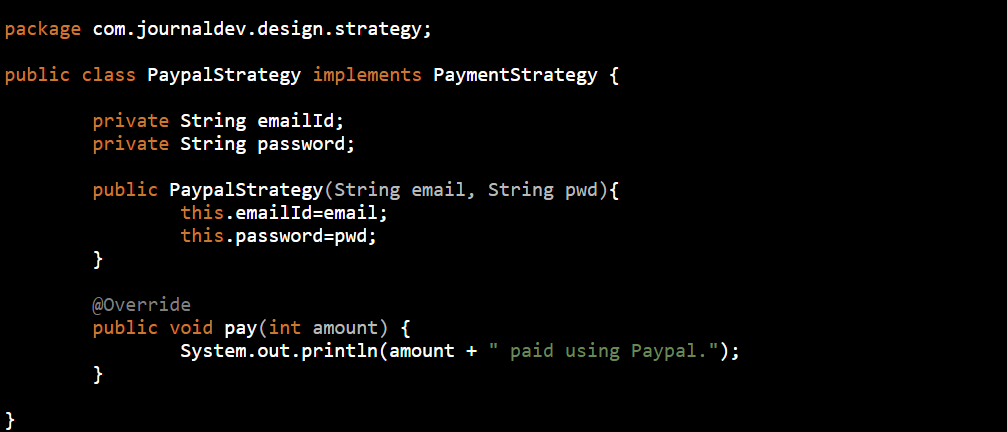


* 1. Une interface **PaymentStrategy**

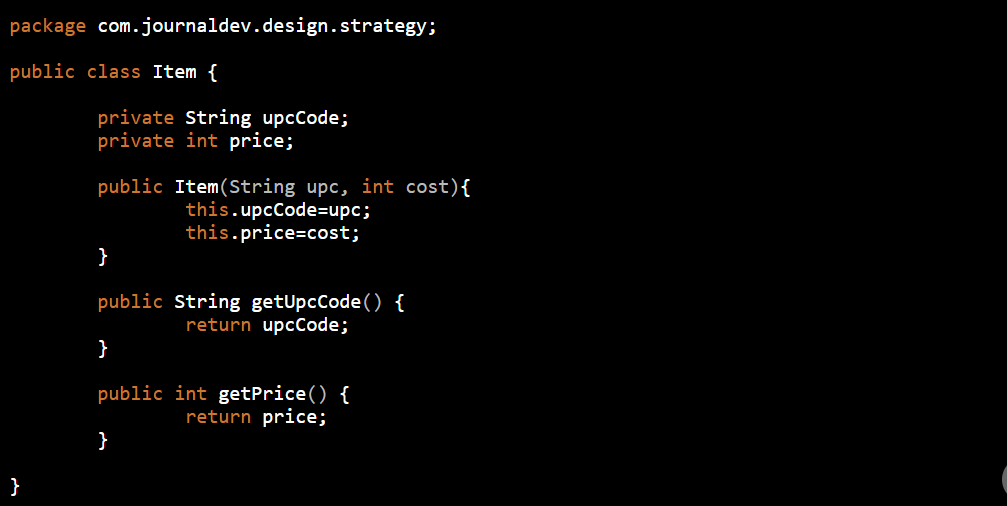


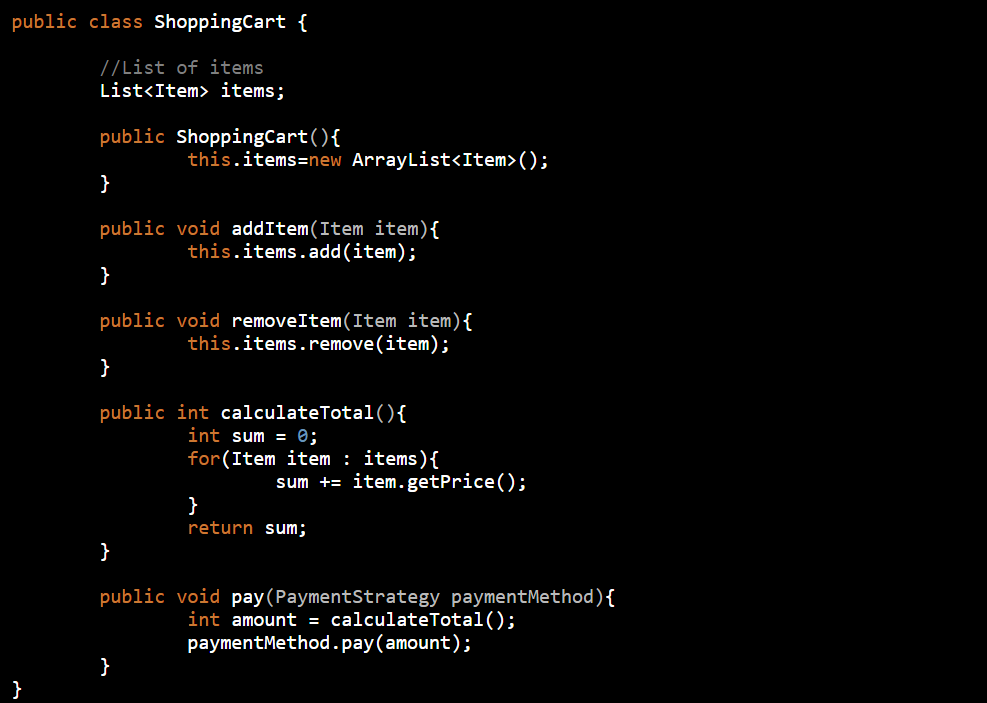
* 1. Deux classes **CreditCardStrategy** et **PayPalStrategy** implémentant l’interface **PaymentStrategy**



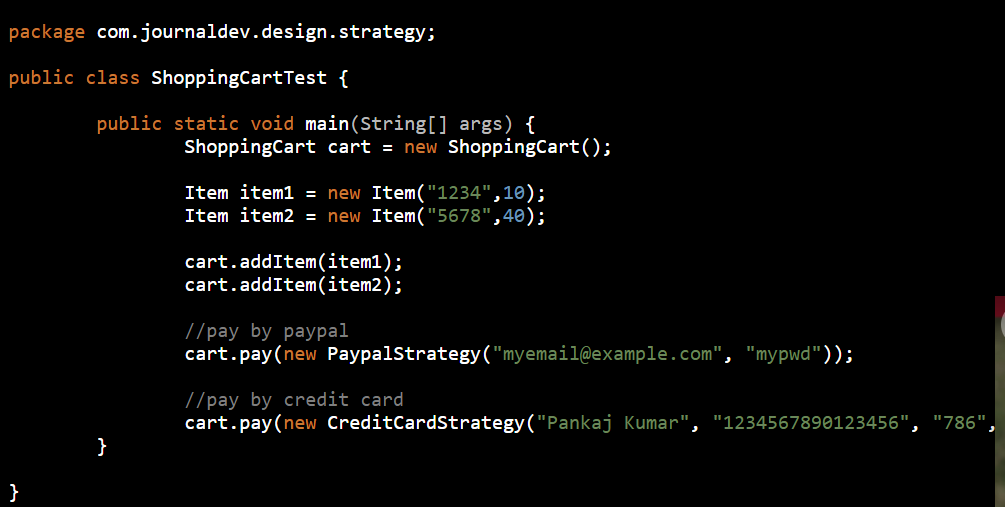


* 1. Un bean **Item**
  2. Une classe **ShoppingCart** implémentant une méthode de paiement avec un input de type **MappingStrategy**





* 1. Une classe de test **ShoppingCartTest**



**Projet MGA**

@Singleton

**public** **class** FichierPlanAuditMapper **extends** AbstractMapper<FichierPlanAudit, FichierPlanAuditEntity> {

/\*\*

\* Instantiates a new plan audit mapper.

\*/

**public** FichierPlanAuditMapper() {

}

/\*\*

\* Inits the mapper.

\*/

@PostConstruct

**public** **void** init() {

setMappingStrategy(LazyFichierPlanAuditStrategy.**class**);

}

}

**public** **abstract** **class** AbstractMapper<M, E> **implements** Mapper<M, E> {

/\*\* The available mapping strategies. \*/

@Inject

**private** Instance<MappingStrategy<M, E>> availableMappingStrategies;

/\*\* The mapping startegy. \*/

**private** MappingStrategy<M, E> mappingStartegy;

/\*\*

\* Invoked by concrete mappers.

\*/

**public** AbstractMapper() {

}

/\*\*

\*

\* inits the concrete extending mapper. annotated with **@PostConstruct**

\*/

**public** **abstract** **void** init();

@Override

**public** E toEntity(M model) {

**return** mappingStartegy.toEntity(model);

}

@Override

**public** M toModel(E entity) {

**return** mappingStartegy.toModel(entity);

}

@Override

**public** **void** setMappingStrategy(Class<? **extends** MappingStrategy<M, E>> mappingStrategyClass) {

**for** (MappingStrategy<M, E> mappingStrategy0 : availableMappingStrategies) {

**if** (mappingStrategy0.getClass().equals(mappingStrategyClass)) {

mappingStartegy = mappingStrategy0;

}

}

}

}

La classe abstraite **AbstractMapper** est appellee avec la méthode **setMappingStrategy** de toutes les classes utilisant une strategie Lazy ou Eager/

### Strategy Design Pattern avec Java 8

Création d’une interface fonctionnelle en utilisant des méthodes statiques.

Ces méthodes ne sont donc pas à implémenter dans des classes

Tout est factorisé dans une seule et même interface :

//Functional interface car :

//Functional because BigDecimal has only one abstract non-Object method

**public** **interface** Discounter {

BigDecimal applyDiscount(BigDecimal amount);

**boolean** equals(Object obj);

//Remplace la méthode applyDiscount dans les classes d'implementations EasterDiscounter et ChrismtasDiscounter

**static** Discounter christmasDiscounter() {

**return** amount -> amount.multiply(BigDecimal.*valueOf*(0.9));

}

**static** Discounter newYearDiscounter() {

**return** amount -> amount.multiply(BigDecimal.*valueOf*(0.8));

}

**static** Discounter easterDiscounter() {

**return** amount -> amount.multiply(BigDecimal.*valueOf*(0.5));

}

}

Appel de l’interface fonctionnelle pour une stratégie :

**public** **class** CallDiscounterFunctionalInterface **implements** Discounter {

**public** **void** call() {

if …

Discounter.*easterDiscounter*();

Else if …

Discounter.*christmasDiscounter*();

Else if …

Discounter.*newYearDiscounter*();

}

}

### Thread Safe

Doc : <https://medium.com/@cancerian0684/singleton-design-pattern-and-how-to-make-it-thread-safe-b207c0e7e368>

« Thread Safe » permet de définir la notion de lock lorsque l’instanciation de la classe **Foo** avec la méthode **getHelper** est réalisée par plusieurs thread en même temps.

### Instanciation single thread

Deux objets peuvent être créés en même temps ou l’un des deux peut être incomplet.

// Single-threaded version  
**class** **Foo** {  
**private** Helper helper;  
  
**public** Helper getHelper() {  
**if** (helper == null) {  
helper = **new** Helper();  
}  
**return** helper;  
}  
  
// other functions and members...  
}

**Avantages**

* + Thread safe sans synchronization
  + Facile à implémenter

**Inconvénients**

* Les deux threads essayent de créer l’objet en même temps, menant à deux objets
* Un thread peut amener à une référence d’objet partiellement initialisé

### Créer la variable d’instance au temps de chargement de la page.

On utilise le mot clé **synchronized**. Le comportement du singleton est prévisible dans un environnement multi thread

// Correct but possibly expensive multithreaded version  
**class** **Foo** {  
 **private** Helper helper;  
 **public** **synchronized** Helper getHelper() {  
 **if** (helper == null) {  
 helper = **new** Helper();  
 }  
 **return** helper;  
 } // other functions and members...  
}

**Avantages**

* + Approche prévisible pour un environnement multithread
  + Lazy initialization

**Inconvénients :**

* + Création d’une ressource qui pourrait ne pas être utilisée dans l’application.
  + **Problème de locking**, décroissement des performances sur un facteur 100
  + L’application client ne passe aucun argument dans le constructeur donc ne peut pas le réutiliser. Désavantage pour la connection à une base de données par exemple

### Optimisation de la synchronisation

**class** **Foo** {  
**private** **volatile** Helper helper;  
  
**public** Helper getHelper() {  
 Helper localRef = helper;  
 **if** (localRef == null) {  
 **synchronized** (this) {  
 localRef = helper;  
 **if** (localRef == null) {  
 helper = localRef = **new** Helper();  
 }  
 }  
}  
**return** localRef;  
}  
  
// other functions and members...  
}

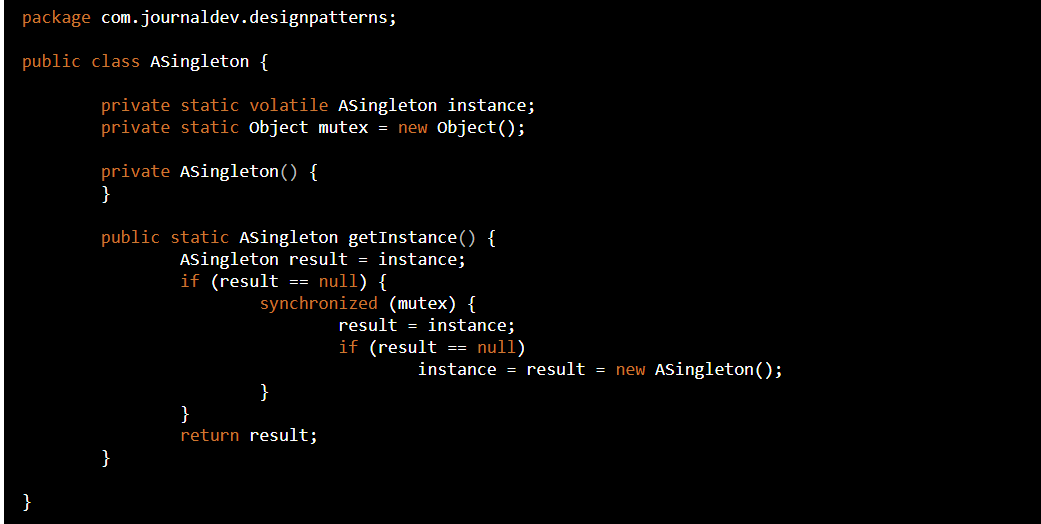
### Utiliser le bloc synchronized à l'intérieur de la boucle conditionnée et avec la variable volatile

**Avantages :**

* + Thread safe
  + L’application peut passer des arguments
  + Lazy initialization
  + Synchronization minimale et applicable uniquement aux premiers chargement des threads quand la variable est nulle

**Inconvénients :**

* + Condition supplémentaires



### **Behavioural design pattern**

### **Design Pattern State**

State est un design pattern permettant de changer le comportement quand son état interne change. Le pattern extrait l’état lié au comportement en différents états dans des sous classes, et force l’objet original à déléguer le traitement.

Permet de réduire le volume d’une seule et même longue méthode, réalisant plusieurs traitements successifs.

1. Classe abstraite

**public** **abstract** **class** EtapesAbstract {

**public** **abstract** **void** treatment();

**protected** **abstract** **void** next(Etape etape);

**protected** **abstract** **void** previous(Etape etape);

}

1. Sous classes

**public** **class** Etape1 **extends** EtapesAbstract{

**public** **void** treatment() {

System.***out***.println("On passe dans l'étape 1");

}

**public** **void** next(Etape etape) {

etape.setState(**new** Etape2());

}

**public** **void** previous(Etape etape) {

System.***out***.println("Pas d'étape précédente");

}

}

1. Classe indépendante

**public** **class** Etape {

**private** EtapesAbstract state = **new** Etape1();

**public** **void** setState(EtapesAbstract state) {

**this**.state = state;

}

**public** **void** setPrevious() {

getState().previous(**this**);

}

**public** **void** setNext() {

getState().next(**this**);

}

**public** EtapesAbstract getState() {

**return** state;

}

}

1. Classe d’implémentation

Etape etape = **new** Etape();

etape.setState(**new** Etape1());

etape.getState().treatment();

etape.setNext();

etape.setState(**new** Etape2());

etape.getState().treatment();

# **Cracking the code interview**

<Https://medium.com/free-code-camp/the-ultimate-guide-to-preparing-for-the-coding-interview-183251ee36c9>

## **Définition : Hash Map**

Un objet HashMap compartimente les éléments à stocker sous forme de paire clé-valeur.

Lorsqu’une paie est insérée dans l’objet, la table de hachage examine le code de hachage de la clé et stocke la paie dans le compartiment possédant le code de la clé utilisée.

Lorsque’on recherche une valeur dans la table de hachage en lui attribuant une clé, il examinera d’abord :

* **Le code de hachage de la clé**
* **Puis le compartiment correspondant**
* **Il comparera ensuite la clé avec toutes les clés des paires stockées dans le compartiment correspondant.**

**Si deux clés sont idéntiques la méthode equals renvoie true et la méthode hashcode renvoie le même nombre. Si les deux clés violent cela, les deux clés pourront être stockées dans des compartiments différents et la table de hachage ne pourra pas trouver les paires clés valeurs.**

**Si deux clés sont différentes, peu importe si leur code de hachage sont identiques ou non, ils seront stockés dans le même compartiment si leur code est identique. La table de hachage utilisera equals pour les différencier.**

* **Contrat entre hashcode et equals**

**Si deux objets sont égaux selon la méthode equals(), leurs code de hachage doivent être identiques. Si deux objets ne sont pas égaux selon la méthode equals(), leur code de hachage peut être identique ou différent.**

## Si le choix de ne pas redéfinir equals est retenu, l'implémentation d'Object ne fait que comparer les adresses, soit le même comportement que ==.

## **Définition : Hash Table**

1 – Récupération de la clé du hash code

2 -Map la clé à un index

3 – L’index route vers des paires clé / valeurs

Le nombre de collisions est très haut **pour retourner les paires Clé / Valeur** : **O(N)** avec N nombre de clés. Une bonne implémentation entraîne la **recherche des clés à un risque de collisions O(1).**

## **Définition : Array Lists**

**public** ArrayList<String> merge(String[] words, String[] more){

ArrayList<String> sentence = **new** ArrayList<String>();

**for** (String w : words) sentence.add(w);

**for** (String w : more) sentence.add(w);

**return** sentence;

}

Les Array Lists ont une taille fixe en Java contrairement à d’autres langages. Pour avoir un dimensionnement dynamique, utiliser les ArrayLists.

Dès qu’un ArrayList est **complet**, il double en taille et la **complexité est de O(N)** en terme de temps mais **l’insertion** est toujours **en O(1)**

Facteur de redimensionnement (**resizing factor**) pour un Array List en Java : **2**

**TODO :**

Pourquoi le temps de d’insertion dans un ArrayList est O(1) ?

Pour N éléments, le temps d’insertion est 1 + ½ + … + N/4 + N/2~ N

Donc pour 1 élément : O(1)

## **Définition : Array String**

**La concaténation d’un string avec un autre possède une complexité en O(N²)**

1+2+4+…+n = n(n+1)/2 ~ O(N²)

**public** String joinWords(String[] words) {

String sentence = "";

**for** (String w : words) {

sentence = sentence + w;

}

**return** sentence;

### }

## **Définition : String Builder**

StringBuilder crée un resizable array, copiant les string en string juste que quand nécesaire

**public** String joinWordsWithStringBuilder(String[] words) {

StringBuilder sentence = **new** StringBuilder();

**for** (String w : words) {

sentence.append(w);

}

**return** sentence.toString();

}

### Arrays and Strings : Probleme  : String avec characters uniques

Demander si le string est un ASCII string ou un UniCode string.

Si pas ASCII demander d’augmenter la storage size

**public** **boolean** isUniqueChars(String str) {

**if** (str.length() > 128) **return** **false**;

**boolean**[] char\_set = **new** **boolean**[128];

**for** ( **int** i =0; i < str.length();i++)

{

**int** val = str.charAt(i);

**if**(char\_set[val]) {

**return** **false**;

}

char\_set[val] = **true**;

}

**return** **true**;

}

Algorithme de complexité O(n) où n est la longueur de la String

Algorithme de complexité O(1) en terme d’espace puisqu’on itère dans la boucle pas plus de 128 fois.

Si la taille de charactère n’est pas fixée :

Complexité O© en espace

Complexité O© ou O(min(c,n)) où c est la taille du caractère set

Diviser la complexité par 8 en utilisant un vecteur de bit

//Diviser la complexité par 8

//Si on utilise que des lower cases de a à z

**public** **boolean** isUniqueChars\_BitVector(String str) {

**int** checker = 0;

**for** ( **int** i =0; i < str.length();i++)

{

**int** val = str.charAt(i) - 'a';

**if**((checker & (1 << val)) > 0) {

**return** **false**;

}

checker |= (1 << val);

}

**return** **true**;

}

### Array and Strings : Probleme : Permutation d’une String

Ecrire une méthode pour vérifier qu’une string est la permutation d’une autre

* Demander si case sensitive
* Demander si les espaces sont à prendre en compte

/ !\ int contient la valeur ASCII de char

/ !\ On historise, dans un tableau de int, la configuration des char de chaque string

**public** **class** CheckPermutation {

String sort(String s) {

**char**[] content = s.toCharArray();

java.util.Arrays.*sort*(content);

**return** **new** String(content);

}

**boolean** permute(String s, String t) {

**if**(s.length() != t.length()) {

**return** **false**;

}

**return** sort(s).equals(sort(t));

}

**boolean** permute\_countNumberCaracters(String s, String t) {

**if**(s.length() != t.length()) {

**return** **false**;

}

**int**[] letters = **new** **int**[128]; //assumption

**char**[] s\_array = s.toCharArray();

**for**(**char** c : s\_array) {

letters[c]++;

}

**for**(**int** i = 0; i < t.length();i++) {

**int** c = (**int**) t.charAt(i);

letters[c]--;

**if**(letters[c] <0)

{

**return** **false**;

}

}

**return** **true**;

}

}

### **Arrays and Strings : Probleme : Urlify**

**Ecrire une méthode permettant de remplacer les espaces par %20**

**On modifie la string en commencant par la fin car il y a un extra buffer a la fin ce qui nous permet de changer**

**les caractères sans se demander ce qu'on écrase**

**2 scans :**

**1 scan pour compter le nombre d'espaces. En multipliant par trois ce nombre, on obtient la longueur à utiliser.**

**1 scan en ordre inverse pour editer la string et remplacer les espaces par %20**

**public** **class** Urlify {

**void** replaceSpaces(**char**[] str, **int** trueLength) {

**int** spaceCount = 0, index, i = 0;

**for**(i=0;i<trueLength;i++) {

**if** (str[i] == ' ') {

spaceCount ++;

}

}

index = trueLength + spaceCount \* 2;

**if**(trueLength < str.length) str[trueLength] = '\0';

**for**(i = trueLength - 1; i >= 0; i --) {

**if** (str[i] == ' ') {

str[index - 1] = '0';

str[index - 2] = '2';

str[index - 3] = '%';

index = index - 3;

}

**else** {

str[index - 1] = str[i];

index --;

}

}

}

}

### **Array and Strings : Probleme : Permutation d’un palindrome**

/\*

\* A partir d'une string ecrire une fonction permettant de dire si c'est une permutation de palindrome

\* Note : Un palindrome est un mot qui peut se lire en avant et en arrière

\* Besoin : avoir le meme nombre de caractère, la moitie pourra etre lu en avant et l'autre en arriere.

\* Seul le caractère au milieu pourra etre lu une seule fois

\* ex : tactcaopapa est un palindrome car 2t 4a 2p 2c et 1o

\*

\* Pour les mots de caractères pairs : tous les caractères doivent etre pairs

\* Pour les mots de caractères impairs : ne doivent avoir qu'un seul caractère impair

\*

\* Donc pour qu'une string soit une permutation d'un palindrome, elle ne peut avoir qu'un seul caractère qui est impair

\*/

**Complexité importante en temps** pour la solution 1 et 2 car on est obligé de compter le nombre de caractères.

**public** **class** PalindromePermutation {

//Solution 1

**boolean** isPermutationOfPalindrome\_solution1(String phrase) {

**int**[] table = buildCharFrequencyTable(phrase);

**return** checkMaxOneOdd(table);

}

//Check that no more one caractere has an odd count

**boolean** checkMaxOneOdd(**int**[] table) {

**boolean** foundOdd = **false**;

**for** (**int** count : table) {

**if** (count % 2 == 1) {

**if**(foundOdd) {

**return** **false**;

}

foundOdd = **true**;

}

}

**return** **true**;

}

//Map each caractere to a number a->0, b->1, c->2 ...

//Non letter characters map to -1

**int** getNumericChar(Character c) {

**int** val = Character.*getNumericValue*(c);

**int** a = Character.*getNumericValue*('a');

**int** z = Character.*getNumericValue*('z');

**if**(val < z && val > a) {

**return** val - a;

}

**return** -1;

}

//Count how many times each character appears

**int**[] buildCharFrequencyTable(String phrase) {

**int**[] table = **new** **int**[Character.*getNumericValue*('z') - Character.*getNumericValue*('a') + 1];

**for** (**char** c : phrase.toCharArray()) {

**int** x = getNumericChar(c);

**if**(x != -1) {

table[x]++;

}

}

**return** table;

}

//Solution 2

**boolean** isPermutationOfPalindrome\_solution2(String phrase) {

**int** countOdd = 0;

**int**[] table = **new** **int**[Character.*getNumericValue*('z') - Character.*getNumericValue*('a') + 1];

**for** (**char** c : phrase.toCharArray()) {

**int** x = getNumericChar(c);

**if**(x != -1) {

table[x]++;

**if**(table[x]%2 ==1) {

countOdd++;

}

**else** {

countOdd--;

}

}

}

**return** countOdd <= 1;

}

}

**Solution 3 : pas besoin de compter le nombre de caractères, on a juste besoin de savoir si le décompte est pair ou impair.**

**De cette manière, on peut utiliser un entier comme un Bit Vector.**

**Quand on voit une lettre, on la map à un entier entre 0 et 26. Ensuite on redirige le bit sur cette valeur.**

**A la fin de l’itération, on verifie qu’au plus un bit dans l’entier est à 1**

**On peut facilement vérifier qu’aucun bit dans l’entier n’est à 1 : comparer l’entier à 0**

**Autre manière : Pour vérifier dans un entier qu’un bit est mis à 1 :**

**Pour un entier de type 00010000 si on soustrait 1 à cet entier = 00001111.**

**Si ensuite on le AND avec l’entier original, on obtient 0 (00010000 & 00001111 = 0)**

//Solution 3

**boolean** isPermutationOfPalindrome(String phrase) {

**int** bitVector = createBitVector(phrase);

**return** bitVector == 0 || checkExactlyOneBitSet(bitVector);

}

//Create the bit vector pour la string. Pour chaque lettre avec une valeur i, la basculer sur le ième bit

**int** createBitVector(String phrase) {

**int** bitVector = 0;

**for**(**char** c : phrase.toCharArray()) {

**int** x = getCharNumber(c);

bitVector = *toggle*(bitVector,x);

}

**return** bitVector;

}

**static** //Toggle the ith bit in the Bit Vector

//ex : bitVector = 1010, index = 1

**int** toggle(**int** bitVector, **int** index) {

//Avoir un bit vector avec le ième bit mis à 1 :

**int** mask = 1 << index;

//Ex mask = 10 (mis à 1 du 2eme bit)

//Ex (bitVector & mask) = 10

**if** ((bitVector & mask) == 0) {

//Ajout d'un flag

bitVector |= mask;

binaireBitVector = Integer.*toBinaryString*(bitVector);

}

**else** {

//Ajout d'un flag

bitVector &= ~mask;

binaireBitVector = Integer.*toBinaryString*(bitVector);

}

**return** bitVector;

}

//Check that exactly one bit is set by substracting 1 depuis l'entier et ANDing it with the original integer

**boolean** checkExactlyOneBitSet(**int** bitVector) {

**return** (bitVector & (bitVector - 1)) == 0;

}

}

Retour : bitVector=10 binaire BitVector=1010 index=1 binaire BitIndex=1

mask : le ieme bit est mis à 1 =2 binaire mask=10

Condition (bitVector & mask)=2 binaire (bitVector & mask)=10

bitVector &= mask :8 binaire bitVector=1000

### **LinkedLists**

Une linkedList est une structure de données représentant une séquence de node.

Comme un array, une linked list ne donne pas un accès constant à un index. Pour trouver le K-ieme element de la liste, il faut itérer sur K éléments.

### Création et suppression d’une linked list

**public** **class** Node {

Node next = **null**;

**int** data;

**public** Node (**int** d ) {

data = d;

}

**void** appendToTail(**int** d) {

Node end = **new** Node(d);

Node n = **this**;

**while** (n.next != **null**) {

n = n.next;

}

n.next = end;

}

Node deleteNode(Node head, **int** d) {

Node n = head;

**if**(n.data == d) {

**return** head.next;

}

**while** (n.next != **null**) {

**if**(n.next.data == d) {

n.next = n.next.next;

**return** head;

}

n=n.next;

}

**return** head;

}

}

/ !\ Runner technique : itérer avec deux pointers en même temps / !\

Utiliser un pointer (fast pointer) qui bouge tous les deux éléments, pour chaque mouvement que p2 fait. Quand p1 arrive à la fin de la linked list, p2 en sera au milieu. Ensuite, bouger p1 en arrière au début et commencer à « balayer » les éléments. Sur chaque itération p2, sélectionne un élément et l’insert après p1.

/ !\ Problèmes récursifs / !\

O(n) en complexité

### Enlever les duplications

### Partitionner une linkedLIst

/\*

\* Créer une partition d'une linkedList, tous les noeuds dont la valeur est inférieur à la partition sont placés en tête de liste

\* Tous les noeuds dont la valeur est supérieure à la partition sont placés en fin de liste

\* La position de la partition n'a pas d'importance

\*

\* Attention : le déplacement d'élements dans une Array est très couteux

\* Avec les linked list : on crée deux linked list, une pour les elements < partition et une pour les elements > ou = à la partition

\* On itere dans la linked list initiale et on ajoute des elements dans la liste pour les elements avant et les elements apres

\* Quand on a finit de parcourire la liste initial, on merge les deux listes

\*/

**public** **class** Partition {

LinkedListNode partition(LinkedListNode node, **int** partitionValue) {

LinkedListNode beforeStart = **null**;

LinkedListNode afterStart = **null**;

LinkedListNode beforeEnd = **null**;

LinkedListNode afterEnd = **null**;

//Partition List

**while**(node != **null**) {

LinkedListNode next = node.next;

node.next = **null**;

**if**(node.data<partitionValue) {

//Insert node into end of before list

**if**(beforeStart == **null**) {

beforeStart = node;

beforeEnd = beforeStart;

}

**else** {

beforeEnd.next = node;

beforeEnd = node;

}

}

**else** {

//Insert node into end of after list

**if**(afterStart == **null**) {

afterStart = node;

afterEnd = afterStart;

}

**else** {

afterEnd.next = node;

afterEnd = node;

}

}

node = next;

**if**(beforeStart == **null**) {

**return** afterStart;

}

//Merge before list and after list

beforeEnd.next = afterStart;

**return** beforeStart;

}

}

}

Version plus courte, pas besoin d’utiliser les 4 variables du code précedent

LinkedListNode partition(LinkedListNode node, **int** x) {

LinkedListNode tail = node;

LinkedListNode head = node;

**while**(node!=**null**) {

LinkedListNode next = node.next;

**if**(node.data < x) {

//Insert node at head

node.next = node;

node = head;

} **else** {

//Insert node at tail

tail.next = node;

tail = node;

}

tail.next = **null**;

//The head has changed so we return it to the user

**return** head;

}

}

### **Threads and Locks**

**Différence entre Thread et Process :**

**Un process peut être considéré comme une instance de programme d’exécution.**

**Chaque process est exécuté dans un espace d’adresse séparé et un process ne peut pas accéder aux adresses et variables d’un autre process.**

**Un thread est une particulière exécution path of a process. Si des threads utilisent les ressources d’un process, ils partageront la même heap space. Les process entre eux, ne pourront pas accéder à la mémoire d’autres process**

### **Problème 1 : Les philosophes**

**Une bande de philosophes est assis autour d’une table. Avec dans chaque main une bière pour boire. Pour boire, ils prennent d’abord celle de gauche avant celle de droite. Un deadlock pourrait probablement arriver si tous les philosophes accèdent à celle de gauche en même temps.**

**En utilisant les threads et locks, implémenter une solution qui préviennent d’un dead lock**

**public** **class** Philosopher **extends** Thread {

**private** **int** bites = 10;

**private** Chopstick left;

**private** Chopstick right;

**public** Philosopher(Chopstick left, Chopstick right) {

**this**.left = left;

**this**.right = right;

}

**private** **void** eat() {

pickUp();

chew();

putDown();

}

**private** **void** putDown() {

right.putdown();

left.putdown();

}

**private** **void** pickUp() {

right.pickup();

left.pickup();

}

**private** **void** chew() {

//

}

**public** **void** run() {

**for** (**int** i = 0 ; i < bites; i++) {

eat();

}

}

}

// => mène à un dead lock et dans le thread utilisé tous les philosophes ont utilisé la bière de gauche et attendent la bière de droite

// Solution 1, les philosophes posent la bière de gauche si ils ne sont pas capables de prendre celle de droite

**public** **class** Philosopher\_solution1 **extends** Thread{

**private** **int** bites = 10;

**private** Chopstick\_solution1 left;

**private** Chopstick\_solution1 right;

**private** **boolean** pickup() {

//Attempt to pick up

**if**(!(left.pickup())) {

**return** **false**;

}

**if**(!right.pickup()) {

left.putdown();

**return** **false**;

}

**return** **true**;

}

**public** **void** eat() {

**if**(pickup()) {

chew();

putdown();

}

}

**private** **void** chew() {

//

}

**private** **void** putdown() {

right.putdown();

left.putdown();

}

}

//If we can't pick up the right chopstick, we release the left one

// Solution 2 : Prioritize the chop sticks with a label from 0 to N. Each philosophes will try to get on his left hand the lowered number chopstick first.

// This will mean each philosophers will try to get the left one before the right one

**public** **class** Philosopher\_solution2 **extends** Thread{

**private** **int** bites = 10;

**private** Chopstick\_solution2 lower;

**private** Chopstick\_solution2 higher;

**private** **int** index;

**public** Philosopher\_solution2(**int** i, Chopstick\_solution2 left, Chopstick\_solution2 right) {

index = i;

**if**(lower.getNumber() <= higher.getNumber()) {

**this**.lower = left;

**this**.higher = right;

} **else** {

**this**.lower = right;

**this**.higher = left;

}

}

**private** **boolean** pickup() {

//Attempt to pick up

lower.pickup();

higher.pickup();

}

**public** **void** eat() {

pickup();

chew();

putdown();

}

**private** **void** chew() {

//

}

**private** **void** putdown() {

higher.putdown();

lower.putdown();

}

**public** **void** run () {

**for** (**int** i = 0; i < bites; i++) {

eat();

}

}

}

Class Chopstick

**import** java.util.concurrent.locks.Lock;

**import** java.util.concurrent.locks.ReentrantLock;

**public** **class** Chopstick {

**private** Lock lock;

**public** Chopstick() {

lock = **new** ReentrantLock();

}

**public** **void** pickup() {

lock.lock();

}

**public** **void** putdown() {

lock.unlock();

}

}

// Solution 1 : Le philosophe dépose la bière de gauche s'il ne peut pas obtenir la bière de droite

**public** **class** Chopstick\_solution1 {

**private** Lock lock;

**public** Chopstick\_solution1() {

lock = **new** ReentrantLock();

}

**public** **boolean** pickup() {

**return** lock.tryLock();

}

**public** **void** putdown() {

lock.unlock();

}

}

//Solution 2 : Ajout de la méthode getNumber pour identifier chaque ChopStick

**public** **class** Chopstick\_solution2 {

**private** Lock lock;

**private** **int** number;

**public** Chopstick\_solution2(**int** n) {

lock = **new** ReentrantLock();

**this**.number = n;

}

**public** **int** getNumber() {

**return** number;

}

**public** **void** pickup() {

lock.lock();

}

**public** **void** putdown() {

lock.unlock();

}

}

### Probleme 2 : Les semaphores

Une classique contenant 3 méthodes. Ordonnancer l’appel des trois méthodes avec trois threads différents

//Avec le code suivant :

**public** **class** Foo {

**public** Foo() {

//...

}

**public** **void** first() {

// ...

}

**public** **void** second() {

// ...

}

**public** **void** third() {

// ...

}

}

//La même instance de Foo sera passée à trois threads différents appliqués à chaque méthode

// Créer un mecanisme permettant d'appeler le threadA pour la méthode first en premier, puis le thread B pour la méthode second, et le thread C pour la méthode third

//Crée un lock car un thread va exécuter le constructor et d'autres threads vont appeler les methodes first second et third produisant des lock

// et des unlock

**public** **class** FooBad {

**private** Lock lock1;

**private** Lock lock2;

**public** FooBad() {

**try**{

lock1 = **new** ReentrantLock();

lock2 = **new** ReentrantLock();

lock1.lock();

lock2.lock();

}

**catch**(Exception e) {

//...

}

}

**public** **void** first() {

**try**{

// ...

lock1.unlock(); //mark finished with first

}

**catch**(Exception e) {

//...

}

}

**public** **void** second() {

**try**{

lock1.lock();

lock1.unlock(); //check if first is finished totally

//...

lock2.unlock();

}

**catch**(Exception e) {

//...

}

}

**public** **void** third() {

**try**{

lock2.unlock();

lock2.lock();

}

**catch**(Exception e) {

//...

}

}

}

//Utilisation des semaphores

**public** **class** FooGood {

**private** Semaphore sem1;

**private** Semaphore sem2;

**public** FooGood() {

**try**{

sem1 = **new** Semaphore(1);

sem2 = **new** Semaphore(2);

sem1.acquire();

sem2.acquire();

}

**catch**(Exception e) {

//...

}

}

**public** **void** first() {

**try**{

// ...

sem1.release(); //mark finished with first

}

**catch**(Exception e) {

//...

}

}

**public** **void** second() {

**try**{

sem1.acquire();

sem1.release(); //check if first is finished totally

//...

sem2.release();

}

**catch**(Exception e) {

//...

}

}

**public** **void** third() {

**try**{

sem2.acquire();

sem2.release();

}

**catch**(Exception e) {

//...

}

}

}

### Problème 3 : TODO Synchronized methods

### **Searching and sorting**

*Référence :*

[*https://github.com/careercup/CtCI-6th-Edition*](https://github.com/careercup/CtCI-6th-Edition)

[*https://github.com/AnthonyNeu/cracking-the-coding-interview*](https://github.com/AnthonyNeu/cracking-the-coding-interview)

### Algorithmes de tri

* **Bubble Sort Algorithm : Time o(n²), Space o(1)**

Sur un Array<String> :

* Itération paire par paire
* Interversion des deux éléments si le 1er > 2ème
* Le plus petit item « bubbles up »
* **Selection Sort Algorithm : Time o(n²), Space o(1)**

Simple mais peu efficace : Scan linéaire

* Trouver le plus petit élement et le placer au début du tableau en le swappant avec le 2ème
* Trouver le 2ème plus petit etc …
* **Quick Sort Algorithm : Time Complexity o(n(log(n))), Space (o(n))**
* Prend un élément aléatoire
* Partitionnement du tableau, de telle sorte que :
  + Tous les éléments qui sont plus petits que l’élément partitionné

viennent avant que les éléments qui sont plus grands que l’éléments partitionné (Serie d’interversion des éléments à chaque itération)

* Répétition du partitionnement autour d’un élément, le tableau pourra devenir trié
* Si le partitionnement choisi n’est pas une médiane du tableau, le tri pourra être très lent

**public** **class** QuickSortAlgorithm {

**static** **void** quickSort(**int**[] arr, **int** left, **int** right) {

**int** index = *partition*(arr, left, right);

**if**(left < index - 1) { //partition left half

**int** ind = index - 1;

*quickSort*(arr, left, index - 1);

}

**if** (right > index) { // partition right half

*quickSort*(arr, index, right);

}

}

**static** **int** partition (**int**[] arr, **int** left, **int** right) {

**int** pivot = arr[(left + right) / 2];

**while** (left <= right) {

//Find element on left that should be on right

**while** (arr[left] > pivot) {

left++;

}

**while** (arr[right] < pivot) {

//Find elements on right that should be on left

right--;

}

//Swap elements and move left and right indices

**if**(left <= right){

*swap*( arr,left,right);

left++;

right--;

}

}

**return** left;

}

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

**int**[] arr = {1,4,2,8,5,9};

*quickSort*(arr,0, arr.length - 1);

}

**private** **static** **void** swap(**int**[] arr, **int** i, **int** j) {

**int** tmp = arr[i];

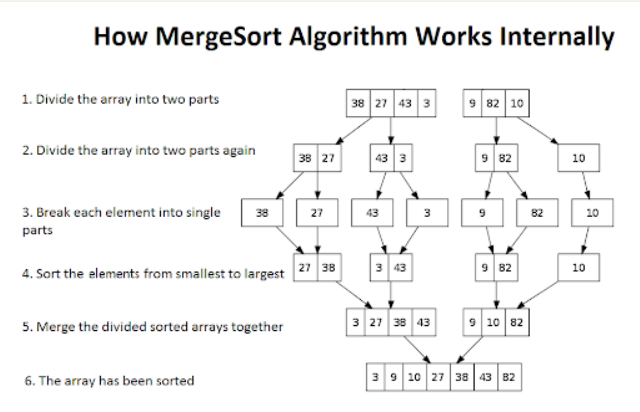
arr[i] = arr[j];

arr[j] = tmp;

}

**Merge sort algorithm : Runtime o(n \* log (n)) , Space o(n)**

* Merge sort divise le tableau en deux.
* Chaque moitié est triée et ensuite fusionnée avec l’autre.
* Chaque moitie a le même algorithme de tri. C’est la fusion (merge) qui fait tout le travail.



Processus de merge :

* Copie tous les éléments du tableau cible « **target** » dans un tableau « **helper** ». Les tableaux « **helperLeft** » et « **helperRight** » gardent une trace où le démarrage du tableau « **left** » et « **right** » devraient être dans le tableau cible « **target** ».
* On itère depuis le tableau « **helper** », en
  + Copiant l’élément le plus petit de chaque moitie de tableau dans le tableau cible « **target** ».
  + Copiant tous les éléments restant dans le tableau cible « **target** »

Seuls les elements restant de la motiie de gauche sont copiés dans le tableau cible car les elements du tableau de droite sont toujours là

Exemple : pour un tableau de type [1,4,5 || 8, 9] ou || indique le point de partition.

Les tableaux helper et target termineront avec [8,9].

Apres avoir copié les elements [1, 4 , 5 et 2] dans le tableau cible, le [8, 9] sera toujours en place dans les deux tableaux.

**public** **class** MergeSortAlgorithm {

**private** **static** **int**[] *array*;

**static** **void** mergesort(**int**[] array) {

**int**[] helper = **new** **int**[array.length];

}

**static** **void** mergesort(**int**[] array, **int**[] helper, **int** low, **int** high) {

**if** (low < high) {

**int** middle = (low + high) / 2;

*mergesort*(array, helper, low, middle); // sort left half

*mergesort*(array, helper, middle + 1, high); // sort right half

*merge*(array, helper, low, middle, high); // merge them

}

}

**static** **void** merge(**int**[] array, **int**[] helper, **int** low, **int** middle, **int** high) {

/\* Copy both halves into a helper array \*/

**for** (**int** i = low; i <= high; i++) {

helper[i] = array[i];

}

System.***out***.println("Copie de array dans helper =" + Arrays.*toString*(helper));

**int** helperLeft = low;

**int** helperRight = middle + 1;

**int** current = low;

/\* Iterate through helper array, compare the left and right half, copying back

\* the smaller element from the two halves into the original array \*/

**while** (helperLeft <= middle && helperRight <= high) {

**if**(helper[helperLeft] <= helper[helperRight]) { //if the left element is smaller than right one

array[current] = helper[helperLeft];

helperLeft++;

}

**else** { //if the right element is smaller than left one

array[current] = helper[helperRight];

helperRight++;

}

current++;

}

/\* Copy the left rest of the left side of the array into the target array \*/

**int** remaining = middle - helperLeft;

**for** (**int** i = 0; i < remaining; i ++) {

array[current + i] = helper[helperLeft + i];

}

System.***out***.println("FIN MERGE array=" + Arrays.*toString*(array));

}

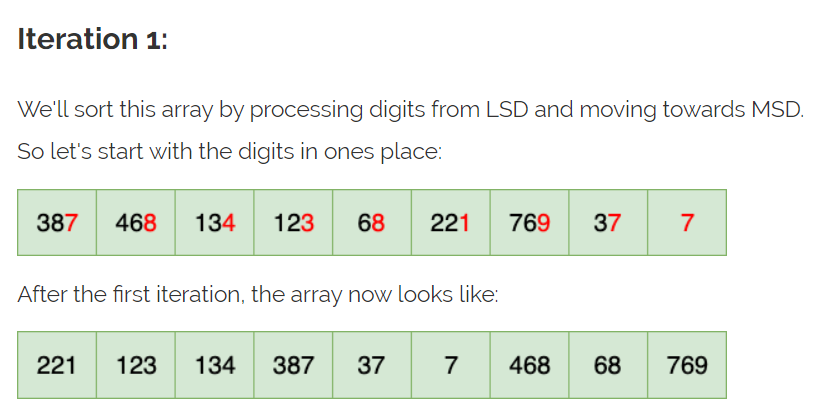
}

**Radix Sort Algorithm (Runtime : O(kn))**

Algorithme de tri utilisé pour les entiers car ils sont définis avec un nombre fini de bits

Sur chaque digit du nombre :

* Tri des éléments en fonction du « last significant digit » (LSD) (ie unité)
* Répétition du procédé pour le digit suivant (ie dizaine etc..)



### Algorithme de recherche

### Binary Search

* Recherche d’un élément x dans le tableau en le comparant en premier au midpoint du tableau
  + Si x < midpoint, on recherche dans la moitie de gauche du tableau
  + Si x > midpoint, on recherche dan la moitié de droite du tableau
* Répétition du procédé jusqu’à ce qu’on trouve x ou que la taille du sous tableau = 0

**public** **class** BinarySearch {

**static** **int** binarySearch(**int**[] array, **int** x)

{

**int** low = 0;

**int** high = array.length -1;

**int** mid;

**while** (low <= high) {

mid = (low + high) / 2;

**if**(array[mid] < x) {

low = mid + 1;

}

**else** **if** (array[mid] > x) {

high = mid - 1;

}

**else** {

mid = x;

}

**return** mid;

}

**return** -1;

}

**static** **int** binarySearchRecursive(**int**[] array, **int** x, **int** low, **int** high) {

System.***out***.println(" binarySearchRecursive debut");

**if** (low > high) **return** -1; //Error

**int** mid = (low + high) / 2;

System.***out***.println(" binarySearchRecursive mid=" + mid);

**if**(array[mid] < x) **return** *binarySearchRecursive* (array, x, mid + 1, high);

**else** **if**(array[mid] > x) **return** *binarySearchRecursive* (array,x, low, mid - 1);

**else** **return** mid;

}

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

**int**[] array = {1,4,2,5,9};

//binarySearchRecursive(array,2,1,4);

*binarySearch*(array,4);

}

}

**Problèmes**

### Q1 : Sorted Merge

Question :

Avec deux sorted arrays A et B où A un buffer suffisamment grand pour stocker B à la fin.

Ecrire une méthode permettant de faire un merge entre B et A à partir d'A dans un ordre trié

Réponse :

Puisque A a suffisamment d'espace à la fin, on compare les éléments de A et de B et on les insère dans l'ordre

- Jusqu'à ce qu'on ait terminé de crawler tous les elements de A et de B

- Le seul problème est qu'il ne faut pas insérer les éléments au début de A sinon il faudra faire un shift pour libérer de l'espace pour les suivants

C'est mieux d'insérer les éléments à la fin où il y a de l'espace disponible

**public** **class** Q1\_SortedMerge {

**void** merge(**int**[] a, **int**[] b, **int** lastA, **int** lastB) {

**int** indexA = lastA - 1; // Index of last element in array A

**int** indexB = lastB - 1; // Index of last element in array B

**int** indexMerged = lastB + lastA - 1;

/\* Merge A et B starting from the last element in each\*/

**while**(indexB >= 0) {

s//end of A is > than the index last element of B

**if**(indexA >= 0 && a[indexA] > a[indexB]) {

a[indexMerged] = a[indexA]; // Copy elements

indexA--;

}

**else**{

a[indexMerged] = b[indexB]; // Copy elements

indexB--;

}

indexMerged--;

}

}

}

### Q2 : Search Rotated Array

Question :

A partir d’un tableau d’entiers, qui a subi un certain nombre de rotations, écrire l’algorithme permettant de :

* Trouver un élément dans le tableau

*Le tableau a subi des rotations dans un ordre ascendant*

* Input 15,16,17,18,25,1,3,5,7,10
* Output : 4 the index of 25

/\*

\* Solution : utiliser la recherche binaire

\* Dans une recherche binaire, on compare x avec le "midpoint" pour vérifier que x appartient ou non à la partie de gauche ou la partie de droite

\* La complication ici est que le tableau a subi des rotations et donc il peut y avoir un point d'inflection

\* Avec l'exemple de ces deux tableaux :

\* {10, 15, 20, 0, 5}

\* {50, 5, 20,30, 40}

\* Les deux tableaux ont un "midpoint" de 20 mais pour le tableau 1 l'élement 5 se trouve à droite tandis que pour le tableau 2 l'élément 5 se

\* trouve à gauche .

\* Par conséquent, comparer x avec le midpoint est insuffisant

\*

\* Si on va plus loin, , On voit que la moitié recherchée du tableau doit être triée normalement dans un tri croissant. On peut donc voir dans la moitié normalement

\* triée pour déterminer si on cherche dans la moitié de gauche ou de droite

\*

\* Par exemple, si nous cherchons 5 dans le tableau 1 et que nous recherchons à partir de l'élément de gauche (10) et du midpoint 20,

\* on constate qu'il va falloir ordonner la partie de gauche (car 10 < 20) et puisque 5 n'est pas dans la partie de gauche, il va falloir chercher

\* dans la partie de droite.

\*

\* Dans le tableau 2 , on peut voir que puisque 50 > 20, la partie de droite doit être triée normalement.

\* On se positionne au milieu (mid point) (20) et on prend l'élément de droite (40) et on vérifie si l'element recherché n'est pas dans ce segment.

\* L'élément 5 n'y est pas, il faudra donc rechercher dans la partie de gauche.

\*

\* La condition epineuse est si l'élément de gauche et le midpoint sont identiques comme dans le tableau

\* {2 ,2 ,2 ,3 ,4 ,2}

\* Dans ce cas, on peut verifier que l'element tout à droite est différent,

\* Si c'est le cas, dans ce cas on effectue la recherche dans la partie de droite

\* Si ca n'est pas le cas, on effectue la recherche dans les deux moitiés

\*/

**public** **class** Q2\_SearchRotatedArray {

**static** **int** search(**int**[] a, **int** left, **int** right, **int** x) {

**int** mid = (left + right)/2;

**if**(left > right) {

System.***out***.println("On passe la left="+ left + " right=" + right);

**return** -1;

}

**if**(a[mid] == x) {

**return** mid;

}

/\* La moitie de gauche ou de droite doit etre normalement triée,

\* Rechercher quelle moitié est normalement triée

\* Et ensuite, utiliser la partie normalement triée pour effectuer la recherche sur x

\*/

**if**(a[left] < a[mid]) { // La moitie de gauche est normalement triée

**if**(x >= a[left] && x < a[mid]) {

**return** *search*(a,left, mid -1, x); // Recherche de gauche

}

**else** {

**return** *search*(a, mid + 1, right, x); // Recherche de droite

}

}

**else** **if** (a[mid] < a[left]) { // La moitié de droite est normalement triée

**if**(x > a[mid] && x <= a[right]) {

**return** *search*(a, mid + 1, right, x); // Recherche de droite

}

**else** {

**return** *search*(a, left, mid-1,x); // Recherche de gauche

}

}

**else** **if**(a[left] == a[mid]) { // L'element de gauche est égal au midpoint

**if**(a[mid] != a[right]) { // L'element de droite est différent du mid point

**return** *search*(a, mid + 1, right, x); // Recherche à droite

}

**else** { // Autrement, on doit rechercher dans les deux moitiés

**int** result = *search*(a, left, mid - 1, x); // Recherche de gauche

**if**(result == -1) {

**return** *search*(a, mid + 1, right, x); // Recherche de droite;

}

**else** {

**return** result;

}

}

}

System.***out***.println("on passe ici");

**return** -1;

}

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

**int**[] a = {5,15,20,0,10};

**int** res = *search*(a, 0, 4, 1);

System.***out***.println(res);

}

}

/\*

\* Ce code aura une compléxité o(n\*log(n)) si tous les elements sont uniques

\* Avec des dupliqués, ce code a une complexité o(n)

\*/

### Spare search

/\*

\* Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string

\* Example : input ball in array {"at","","","ball","point","","",""} Output : 4

\*

\* S'il n'y avait pas de empty strings, on pourrait utiliser la recherche binaire

\* Avec des empty strings, on puet implémenter une simple modification de la recherche binaire

\* Lors de la comparaison par rapport a mid, si mid est vide, on le bouge a cote de l'element non vide le plus pres

\*/

**public** **class** Q3\_SpareSearch {

**int** search(String[] strings, String str, **int** first, **int** last) {

**if**(last < first) {**return** -1;}

**int** mid = ( first + last ) / 2;

//If mid is empty, find the closest empty strings

**if**((strings[mid]).isEmpty()) {

**int** left = mid - 1;

**int** right = mid + 1;

**while** (**true**) {

**if**(left < first && right > last) {

**return** -1;

}

**else** **if** (right <= last && !strings[right].isEmpty()) {

mid = right;

**break**;

}

**else** **if** (left >= first && !strings[left].isEmpty()) {

mid = left;

**break**;

}

right++;

left--;

}

//Search it recursively to find the string in array

**if**(str.equals(strings[mid])) {

**return** mid;

}

**else** **if** (strings[mid].compareTo(str) < 0) { // Search right

**return** search(strings, str, mid + 1, last);

}

**else** { // Search left

**return** search(strings, str, first, mid - 1);

}

}

}

**int** search(String[] strings, String str) {

**if**(str=="" || str.isEmpty() || strings == **null**) {

**return** -1;

}

**else** {

**return** search(strings, str, 0, strings.length - 1);

}

}

}

/\*

\* Complexité O(n) dans le pire des cas

\* Impossible d'avoir un meilleur algorigthme en terme de complexité. Si le tableau ne possède que des elements vides, sauf pour

\* un element, alors il sera nécessaire de regarder chaque element du tableau.

\*

\* Débattre avec l'interviewer sur la manière de gérer le string empty, doit on le considérer comme une erreur ou on doit recherche la localisation de cette mpty

\* string qui est une O(n) itération

\*/

### Sort big file

Avec un fichier de 20 Gb, avec une string par ligne, expliquer comment on peut trier le fichier

Quand l’interviewer donne une limite de 20 Gb , il suggère qu’il ne faut pas emporter toute la data dans la mémoire.

On divise donc le fichier en « chunks » qui ont x megabytes chaque, où x est le montant de la mémoire disponible. Chaque chunk est trié séparément et sauvé dans le système de fichiers.

Quand chaque chunk est trié, on effectue un merge de chunk, un par un. A la fin on a un fichier complétement trié.

Cet algorithme est connu par le nom de External sort.

### Rank from stream

**Imaginez de lire un flux d'entiers.**

Implémenter un algorithme permettant de chercher **un rang d'un nombre x** (le nombre de valeur inférieur ou égal à x)

Implémenter la structure de données permettant d'effectuer ces opérations :

* Méthode track(int x), appelé quand chaque nombre est généré
* Méthode getRankOfNumber (int x)

qui retourne le nombre de valeurs inférieures ou égales à x, incluant x

**SOLUTION :**

Une solution facile serait d'avoir un tableau qui contient tous les éléments

Quand un élément arrive en entrée : Faire un remplacement des autres éléments pour faire de la place

* Efficace pour **getRankOfNumber**
* Inefficace pour insertion et mise à jour des éléments

**Utiliser un binary search tree**

* Complexité **o(log(n))** en temps, où n est le taille de l'arbre pour insertion des éléments

Trouver le rang des éléments :

* Faire une traversée ordonnée, en gardant un compteur qui permettra de savoir si l'on dépasse le rang de l'element recherché :
  + **Chercher x en se déplaçant sur la gauche** : le compteur ne changera pas : tous les éléments sur la droite sont plus grands que x.
  + **Chercher x en se déplaçant sur la droite**: Compteur++ : les éléments sur la gauche sont inférieures à x
* Au lieu de compter la taille du sous arbre de gauche, on peut garder cette information quand on ajoute des elements à l'arbre.

(Exemple de l'arbre dans le livre)

Chercher le **rang du nœud 24** :

* 24 > Nœud racine 20 : Positionner 24 sur la droite.
* 20 a 4 éléments dans le sous arbre de gauche, et avec le nœud racine
* **Rang de 5 pour le nœud 20**

**Chercher le rang du nœud suivant 25 :**

* 24 doit être donc positionné sur la gauche, le compteur reste inchangé
* **Rang de 5 pour le nœud 25**

**Chercher le rang du nœud suivant 23 :**

* 24 > 23 : Nœud 24 à positionner sur la droite.
* 24 > 23 : Compteur++ : 23 n'a pas de noeud sur la gauche
* **Rang de 6 pour le noeud 24**

Finalement, nous retournons 24 avec un compteur de 6

* Complexité o(log(n)) pour un arbre organisé (noeud < a gauche et noeud > a droite)
* Complexité o(n) pour un arbre "unbalanced"

Si la valeur d n'est pas trouvée dans l'arbre, on cherche la valeur -1 retournée dans tout l'arbre

\* Algorithme récursif de getRank

\* int getRank(Node node, int x) {

\* if x is node.data; return node.leftSize()

\* if x is on left of node, return getRank(node.left, x)

\* if x is on right of node, return getRank(node.left,x) + 1 + getRank(node.right,x)

\* }

// Relatif a Q4\_RankNodeFromStream

**public** **class** RankNode {

**public** RankNode left, right;

**public** **int** data = 0;

**public** **int** left\_size = 0;

**public** RankNode(**int** d) {

data = d;

}

**public** **void** insert(**int** d) {

**if**(d <= data) { // d <= data, on insère à gauche

**if** (left != **null**) {

left.insert(d);

}

**else** left = **new** RankNode(d);

left\_size++;

}

**else** { // d > data, on insère à droite

**if**(right != **null**) {

right.insert(d);

}

**else** right = **new** RankNode(d);

}

}

**public** **int** getRank(**int** d) {

System.***out***.println("d =" + d + " left\_size=" + left\_size);

**if** (d == data) { // La donnée d == à la valeur du noeud, le rang est donc celui du compteur

**return** left\_size;

}

**else** **if** (d < data) { // La donnée à insérer est < à la valeur du noeud, le rang ne change pas par rapport à celui de gauche

**if**(left == **null**) **return** -1;

**else** **return** left.getRank(d);

}

**else** { // La donnée d à insérer est > valeur du noeud, le rang de droite est incrémenté de 1 par rapport au rang du noeud de gauche

**int** right\_rank = right == **null** ? -1 : right.getRank(d);

**if**(right\_rank == -1) **return** -1;

**else** **return** left\_size + 1 + right\_rank;

}

}

}

**public** **class** Q4\_RankFromStream {

RankNode root = **null**;

**void** track(**int** number) {

**if** (root == **null**) {

root = **new** RankNode(number);

}

**else** {

root.insert(number);

}

}

**int** getRankNumber(**int** number) {

**return** root.getRank(number);

}

}

**public** **class** Handler {

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

RankNode root = **new** RankNode(20);

root.insert(15);

root.insert(10);

root.insert(5);

root.insert(13);

root.insert(25);

root.insert(23);

root.insert(24);

System.***out***.println("root" + root);

**int** rank = root.getRank(23);

}

}

### Find Duplicates

Référence :

<https://flylib.com/books/en/2.300.1/bit_vectors.html>1

<https://stackoverflow.com/questions/9141830/explain-the-use-of-a-bit-vector-for-determining-if-all-characters-are-unique>

size >> 5 On décale les bits de size de 5 positions de la gauche vers la droite

1 << i On met à 1 le ième bit

|= est similaire à +=.

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html>

Enoncé :

Avec un tableau de tous les nombres de 1 à N, où N <= 32 0000.

Le tableau peut avoir des dupliqués et on ne sait pas quelle est la valeur du nombre N

Avec seulement 4 KiloBytes de mémoire disponibles, comment peut on écrire tous les dupliqués dans le tableau ?

Réponse :

On a 4 kilobytes de mémoire donc on peut adresser 4 \* 1000 \* 2^10 bits

* En utilisant un vecteur de 32 000 bits, ou chaque bits représente un entier.
* En utilisant ce vecteur de bits, on peut itérer à travers le tableau, en flaguant chaque element v en settant v à 1

Quand on passe au devant d'un element dupliqué, on l'écrit

**public** **class** Q5\_FindDuplicates {

**public** **static** **void** checkDuplicates(**int**[] array) {

System.***out***.println("[Q5\_FindDuplicates] checkDuplicates array=" + Arrays.*toString*(array));

Q5\_BitVector bv = **new** Q5\_BitVector(32000);

**for** (**int** i = 0; i < array.length; i ++) {

**int** num = array[i];

**int** num0 = num - 1; // BitSet starts at 1, array starts at 0

System.***out***.println("[Q5\_FindDuplicates] checkDuplicates num0=" + num0);

**if**(bv.get(num0)) {

System.***out***.println("[Q5\_FindDuplicates] num0=" + num0);

}

**else** {

bv.set(num0);

}

}

}

}

**public** **class** Q5\_BitVector {

**int**[] bitset;

**public** Q5\_BitVector(**int** size) {

**int** a = (size >> 5) + 1;

bitset = **new** **int**[(size >> 5) + 1 ]; // Divide by 32

// décale les bits de size de 5 positions de la gauche vers la droite

System.***out***.println("[Constructeur BitVector] (size)=" + size + "size=" + Integer.*toBinaryString*(size) + "(size>>5) + 1= " + a + " (size >> 5)=" + Integer.*toBinaryString*((size >> 5)));

//System.out.println("[Constructeur BitVector] bitset=" + Arrays.toString(bitset));

}

**public** **void** set(**int** pos) {

**int** wordNumbers = (pos >> 5); // Divide by 32

// Les 5 bits de gauche sont à 1

**int** bitNumbers = (pos & 0x1F); // Mod 32

System.***out***.println("[set] wordNumbers=" + Integer.*toBinaryString*(wordNumbers) + " bitNumbers=" + Integer.*toBinaryString*(bitNumbers));

System.***out***.println("[set] Avant set bitset[wordNumbers]=" + Integer.*toBinaryString*(bitset[wordNumbers]));

bitset[wordNumbers] |= 1 << bitNumbers; // Le bit à l'index BitNumbers est tourné à

//System.out.println("[set] bitset[wordNumbers]=" + Arrays.toString(bitset));

System.***out***.println("[set] Apres set bitset[wordNumbers]=" + Integer.*toBinaryString*(bitset[wordNumbers]));

}

**public** **boolean** get(**int** pos) {

**int** wordNumbers = (pos >> 5); // Divide by 32

**int** bitNumber = (pos & 0x1F);

System.***out***.println("[get] pos=" + Integer.*toBinaryString*(pos) + " wordNumbers=" + Integer.*toBinaryString*(wordNumbers) + " bitNumber=" + Integer.*toBinaryString*(bitNumber));

//System.out.println("[get] bitset=" + Arrays.toString(bitset) + " bitset[wordNumbers]=" + Integer.toBinaryString(bitset[wordNumbers]) + " (1 << bitNumber)=" + Integer.toBinaryString((1 << bitNumber)));

System.***out***.println("[get] bitset[wordNumbers]=" + Integer.*toBinaryString*(bitset[wordNumbers]) + " (1 << bitNumber)=" + Integer.*toBinaryString*((1 << bitNumber)));

**return** (bitset[wordNumbers] & (1 << bitNumber)) != 0;

}

}

### Pics et Vallées

Enoncé :

Dans un tableau d'entier, récupérer un element qui est plus grand égal à l'entier adjacent

et une valley qui est plus petite ou égal à l'entier adjacent.

Par exemple : Dans le tableau [5,8,6,2,3,4,6], [8,6] sont des pics et [5,2] sont des vallées

Dans un tableau d'entier, trier le tableau en une séquence alternante de pics et de vallées

[INPUT] [5,3,1,2,3]

[OUTPUT] [5,1,3,2,3]

Solution non optimale

1. Trier le tableau dans un tri ascendant ou descendant
2. Arranger ensuite le tableau dans une séquence alternée de pics et vallées
3. Après l’étape 1, un tableau trié apparaîtra de la manière suivante :

0 1 4 7 8 9

1. Pour l’étape 2 :
   1. 0 est ok
   2. 1 est à la mauvaise place, on le swap avec le 0 ou le 4, on les wap avec le 0

1 0 4 7 8 9

* 1. .4 est ok
  2. 7 est à la mauvaise place, on le swap soit avec le 4 ou le 8, on choisit le 4

1 0 7 4 8 9

* 1. 9 est à la mauvaise place, on le swap avec le 8

1 0 7 4 9 8

Le tableau initial ne possédant pas de configuration particulière, on peut donc utiliser cette approche sur tout type de tri de tableau

Détail de l’algorithme :

1. Trier le tableau dans un ordre ascendant
2. Itérer sur les éléments, en parant de l’index 1 avec un pas de 2
3. Sur chaque éléments, effectuer un swap à partir de la configuration *small => medium => large*, obtenu avec le tri ascendant, pour obtenir la configuration *medium => small => large*

Compléxité o(n(log(n)))

**public** **class** Q6\_PeaksAndValley {

**public** **void** sortValleyPeak(**int**[] array) {

Arrays.*sort*(array);

**for** (**int** i = 1; i <= array.length;i+=2) {

swap(array,i-1,i);

}

}

**public** **void** swap(**int**[] array, **int** left, **int** right) {

**int** tmp = array[left];

array[left] = array[right];

array[right] = tmp;

}

}

### Notes sur refactoring été 2020 :

**Replace conditionnal with polymorphism**

Classe handler pour déterminer la stratégie sur le type d’implémentation à appliquer

Interface pour regrouper les implementations et le traitement utilisée

La classe handler possède une primitive de type interface

Cf exemple <https://stackoverflow.com/questions/48377525/replace-conditional-with-polymorphism>

Class DiscountFactory {

private static final Map<String, DiscountStrategy> strategies = new HashMap<>();

private static final DiscountStrategy DEFAULT\_STRATEGY = () -> 0;

static {

strategies.put("code1", () -> 10);

strategies.put("code2", () -> 20);

}

public DiscountStrategy getDiscountStrategy(String priceCode) {

if (!strategies.containsKey(priceCode)) {

return DEFAULT\_STRATEGY;

}

return strategies.get(priceCode);

}

}

After that, the Product class can be simplified:

class Product {

private DiscountStrategy discountStrategy;

Product(DiscountStrategy discountStrategy) {

this.discountStrategy = discountStrategy;

}

public int getDiscount() {

return discountStrategy.getDiscount();

}

}

DiscountFactory factory = new DiscountFactory();

Product product = new Product(factory.getDiscountStrategy("code1"));

List Completable Future

1. List<CompletableFuture<Future<Type>>> = Arrays.asList(new CompletableFuture

**Interface fonctionnelle**

* Appeler une interface fonctionnelle
* Créer une interface avec l’annotation @FunctionalInterface qui permet de vérifier que l’interfacec est toujours fonctionnelle en ajoutant des méthodes

@FunctionalInterface

**public** **interface** Creator<T> {

T create(String firstName, String lastName);

}

* Définir une méthode d’implémentation

**public** **class** NameParser<T> {

**public** T parse(String name, Creator creator) {

String[] tokens = name.split(" ");

String firstName = tokens[0];

String lastName = tokens[1];

**return** (T) creator.create(firstName, lastName);

}

}

* La méthode abstraite déclarée dans l’interface fonctionnelle n’a pas besoin d’être définie  
  C’est l’interêt des interfaces fonctionnelles, pas besoin de définir une classe anonyme.
* Appeler la méthode avec une expression lambda

NameParser parser = **new** NameParser();

Name resWithLambda = (Name) parser.parse("Eric Clapton", (s1, s2) -> **new** Name(s1, s2));

System.***out***.println("resWithLambda name=" + resWithLambda.getFirstName() + " res nickname=" + resWithLambda.getLastName());

* Mise à jour d’un objet avec Stream API

Séquence : Stream / map / filter / findAny / setter du bean

**public** **void** updateUser(Integer userid, String username) {

*users*.stream()

.filter(x -> x.getUserid() == userid)

.findAny()

.orElseThrow(() -> **new** RuntimeException("Item not found"))

.setUsername(username);

}

* API Hibernate pour la creation d’un repository JAVA

Annotations de jointures des tables / Creation des beans

* API Stream pour créer une liste
* Interface fonctionnelle Predicate pour la creation de filtres dissocies du parcours du stream API

Exemple :

Predicate<Order> inLeapYear = o -> o.getCreationDate().isLeapYear();

* Création d’un objet Map à partir d’une liste imbriquée par la méthode flatMap et la méthode GroupingBy

**return** orders.stream()

.filter(inLeapYear)

.flatMap(o -> o.getOrderLines().stream())

.collect(*groupingBy*(OrderLine::getProduct, *summingInt*(OrderLine::getItemCount)));

**Utilisation du design pattern null object**

On l’utilise lorsque l’on veut appliquer une méthode sur un objet potentiellement nul (ici Customer)

Avant :

**if** (customer == **null**) {

plan = BillingPlan.basic();

}

**else** {

plan = customer.getPlan();

}

Apres :

**class** **NullCustomer** **extends** Customer {

**boolean** isNull() {

**return** **true**;

}

Plan getPlan() {

**return** **new** NullPlan();

}

*// Some other NULL functionality.*

}

*// Replace null values with Null-object.*

customer = (order.customer != **null**) ?

order.customer : **new** NullCustomer();

*// Use Null-object as if it's normal subclass.*

plan = customer.getPlan();

Ce qu’il faudra faire pour appliquer ce pattern dans la classe associée à MGA

//A faire

// on definit un objet null NullObjectTypeGroupage

// qui definit un type groupage null

//En cours: TypeGroupage represente la classe Customer utilisée sur refactoring Guru

//ensuite a faire pour la classe SQL params regroupant tous les paramètres SQL a utiliser dans la méthode finale

//La condition doit être effectuée sur l'objet sur lequel on applique la méthode

typeGroupage = (calculTauxVO == null) ? calculTauxVO.getTypeGroupage() : **new** NullObjectTypeGroupage();

typegroupage = typeGroupage.getTypeGroupage();

**Convertir une liste imbriquée en Map**

Utiliser la liste initiale

Appeler un getter permettant de retourner une list depuis la liste initiale

Utiliser la méthode groupingBy de l’interface Collect de la manière suivante :

² Map<Integer, List<Etape>> map2 = listMain.stream()

.flatMap(m -> m.getEtapes().stream())

.collect(Collectors.groupingBy(Etape::getId));

**Utilisation design pattern state** sur classe java projet MGA CrbChargementCndtActionImpl

Conditionnement réalisé entre chaque etape sur des variables portées par l’interface chap eautant les états.

**Refactoring from Imperative to Reactive Java 9**

<http://dhaval-shah.com/refactoring-from-imperative-to-reactive-implementation/>

**Quand utiliser la reactive API ?**

A Utiliser si haut niveau de concurrence et si des threads sont a paralleliser

Attention aux chose suivantes :

* Si Equipe volumineuses
* Si code existant synchrone satisfait le besoin

Il vaut mieux ne pas l’utiliser car le debug et la partie développement sera plus difficile, transfert de compéternce

* Si transfert de compétence trop important, le developpeur peut utiliser l’API WebClient

<https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html>

<https://howtodoinjava.com/spring-webflux/webclient-get-post-example/>

Un thread s’exécute en mode NON Blocking IO , ne se bloque pas dans l’attente du retour de l’API Calls.

Une notification est ensuite retournée à l’utilisateur qui a émis l’appel REST pendant ce temps, il peut effectuer d’autres appels et d’autres threads peuvent être exécutées en parallèle.

Besoin potentiel :

* Génération de rapports , l’écran ne se bloque pas l’utilisateur reçoit une notification dès que le rapport est généré.
* Complexes interactions utilisateurs, données qui a besoin d’être rafraichie, ou « convoluted sequences of network calls »

Design Pattern Builder

Simplification de l’utilisation des contructeurs

Declaration d’une interface regroupant les différentes classes appelées pour utiliser les constructeurs

Example :

**Interface Builder**: regroupe les Set utilisées par les différentes classes Builder

**public** **interface** Builder {

**void** setType(Type type);

**void** setSeats(**int** seats);

**void** setEngine(Engine engine);

**void** setTransmission(Transmission transmission);

**void** setTripComputer(TripComputer tripComputer);

**void** setGPSNavigator(GPSNavigator gpsNavigator);

}

**Classe CarBuilder :** création de différents type d’Objet Car

Les getter / setter de CarBuilder ont des types d’objets et non primitifs (exemple : GPSNavigator, Transmission …)

**public** **class** CarBuilder **implements** Builder {

**private** Type type;

**private** **int** seats;

**private** Engine engine;

**private** Transmission transmission;

**private** TripComputer tripComputer;

**private** GPSNavigator gpsNavigator;

…

**public** Car getResult() {

**return** **new** Car(type, seats, engine, transmission, tripComputer, gpsNavigator);

}

**Classe Handler**

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

Director director = **new** Director();

// Director gets the concrete builder object from the client

// (application code). That's because application knows better which

// builder to use to get a specific product.

CarBuilder builder = **new** CarBuilder();

director.constructSportsCar(builder);

// The final product is often retrieved from a builder object, since

// Director is not aware and not dependent on concrete builders and

// products.

Car car = builder.getResult();

System.***out***.println("Car built:\n" + car.getType());

CarManualBuilder manualBuilder = **new** CarManualBuilder();

// Director may know several building recipes.

director.constructSportsCar(manualBuilder);

Manual carManual = manualBuilder.getResult();

System.***out***.println("\nCar manual built:\n" + carManual.print());

}