2019

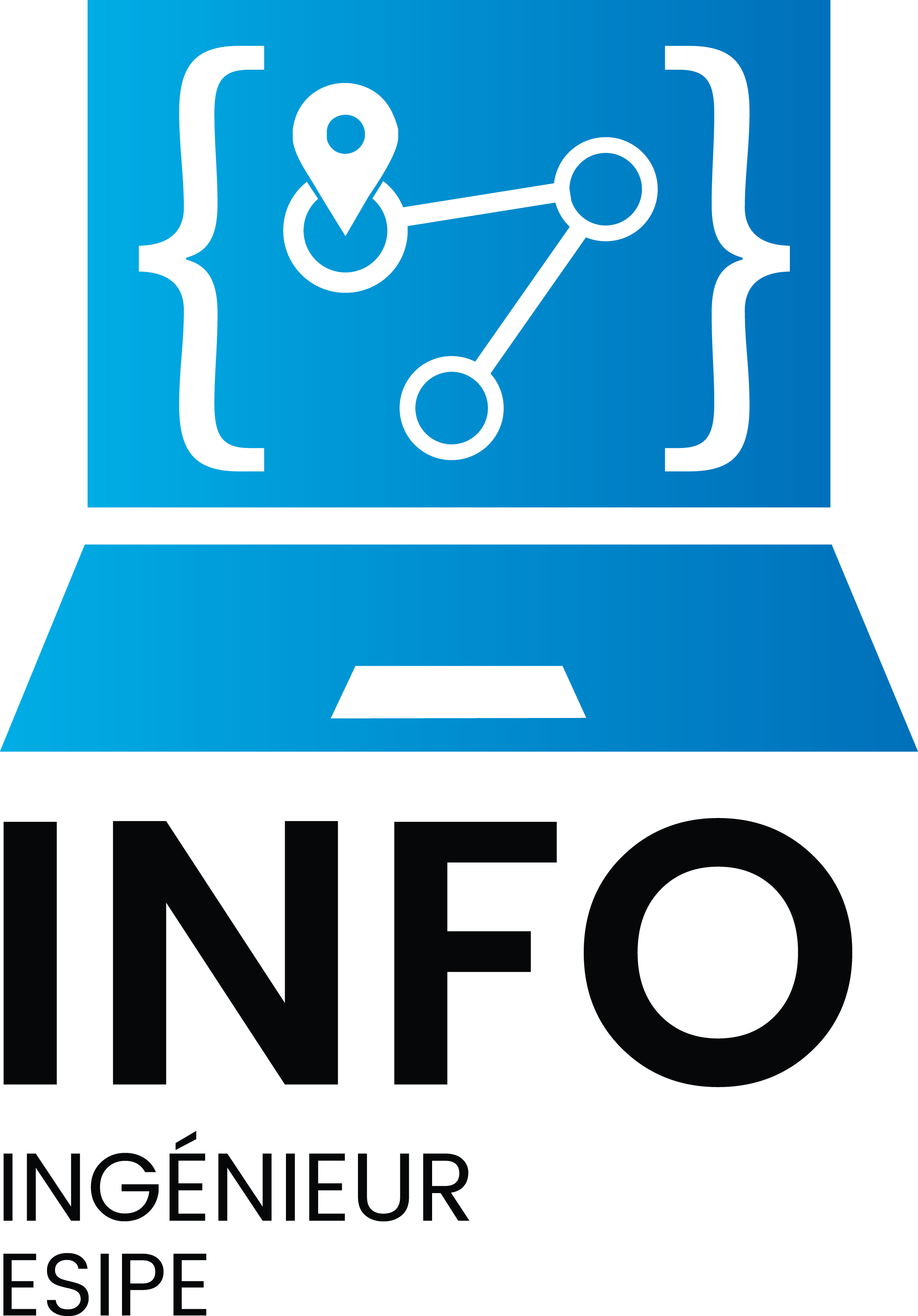
Jonathan CRÉTÉ – Filière Informatique

Compte rendu de TP : Rappel de notions de programmation objet

Java avancé – TP n°2









Compte rendu de TP : Rappel de notions de programmation objet

**Exercice 1 - Path, Stream et try-with-resources**

**1)** Avant de commencer, rappeler pourquoi, depuis la version 7 de Java, on représente un chemin en Java en utilisant la classe java.nio.file.Path et pas la classe java.io.File comme précédemment.

Tout d’abord, le package *java.nio.file.Path* reprend les trois-quarts des fonctionnalités de *java.io.File*. Par exemple les méthodes du package ne levaient pas d’exception en cas de mauvais comportement, ou encore on ne pouvait changer les permissions des fichiers. Il vaut mieux utiliser le package *java.nio.file.Path* que *java.io.File* à cause de l'encodage. En plus la classe java.nio.file.Path facilite pour le développeur le fonctionnement des entrées-sorties. La classe *java.io.File.java.io.File* est destinée à ne plus être utilisée (**deprecated**).

**2)** Créer un Path en utilisant la factory method Path.of() sur le fichier movies.txt.

On crée le Path.

|  |
| --- |
| var path **=** Path**.**of**(**"data/movies.txt"**);** |

**3)** En utilisant la méthode Files.lines qui permet d'extraire toutes les lignes d'un fichier, afficher celles-ci à l'aide de la méthode forEach. Dans un premier temps, nous allons essayer de gérer l'exception éventuelle à l'aide d'un try-catch.

|  |
| --- |
|  |

**4)** En fait, le code que vous avez écrit est (très probablement) faux, car vous avez ouvert un fichier, vu celui-ci comme un Stream mais vous avez oublié de fermer le Stream pour fermer le fichier sous-jacent avec la méthode close.

Modifiez votre code.

Pour développer la méthode en utilisant un objet StringBuilder et sans retour à la ligne final, on utilise la méthode substring :

|  |
| --- |
| public String toString**()** **{**  StringBuilder st **=** **new** StringBuilder**();**    **for(**Car car**:** **this.**cars**)** **{**  st**.**append**(**car**);**  st**.**append**(**"\n"**);**  **}**  **if(!**cars**.**isEmpty**())** **{**    st**.**substring**(**0**,** st**.**length**()-**1**);**  **return** st**.**toString**();**  **}**  **return** ""**;**  **}** |

**5)** Le code est encore (très probablement) faux, car si vous avez une exception lors du forEach, la méthode close ne sera jamais appelée.

Modifiez votre code en utilisant un finally pour résoudre le problème.

On développe la méthode *remove(...)* demandée (**Voir CarRental.java**)

**6)** Si vous vous êtes laissés guider par Eclipse, vous vous êtes probablement donné beacoup de mal pour réussir à fermer un stream qui n'a même pas été ouvert...

Modifiez votre code pour ne pas avoir à initialiser le stream avec null..

L'interface Stream

**7)** Quelle est la différence entre l'utilisation de la construction try-catch et un throws.

Pourquoi vaudrait-il mieux utiliser un throws ici?

Modifier le code en conséquence.

Dans

**8)** En fait, il est plus pratique d'utiliser la construction try-with-resources, le try(...) car dans ce cas l'appel à close est fait automatiquement à la fin du bloc try.

Modifiez une nouvelle fois votre code.

On utilise un **try-with-resources** comme demandé :

|  |
| --- |
| **try(**Stream**<**String**>** st **=** Files**.**lines**(**path**)){**  st**.**forEach**(**System**.**out**::**println**);**  **}** |

**9)** A la maison, faites une recherche pour savoir pourquoi le try(..) est mieux que le try/finally.

Notez que les deux premiers morceaux de code que vous avez écrits semblaient bons et marchaient alors qu'ils étaient farcis de bugs. C'est le gros problème des tests : ce n'est pas parce qu'un test affiche ce qu'il faut que le code est correct.

**Exercice 2 – Movies Stars**

**1)** On cherche dans un premier temps à écrire une méthode actorsByMovie qui prend en paramètre un fichier (un Path) et renvoie une Map qui associe à un nom de film une List des noms des acteurs.

Pour cela, écrivez le code qui lit le fichier ligne à ligne avec la méthode Files.lines et qui stocke les infos dans la Map.

Pour séparer une chaîne de caractères en plusieurs parties, il existe la méthode String.split(). Il existe de plus une méthode Stream.skip() qui permet de ne pas prendre en compte certaines valeurs dans un stream.

Si ce n'est pas déjà fait, allez regarder la documentation de la méthode Collectors.toUnmodifiableMap et utilisez-la dans votre code.

|  |
| --- |
| /\*\*  \* **@param** model  \* **@param** yearBuild  \*/  public Car**(**String model**,** int year**)** **{**  **super(**year**);**  **this.**model **=** Objects**.**requireNonNull**(**model**);**  **}** |

**2)** On cherche à afficher le nombre total d'acteurs ayant joué dans au moins un film.

L'idée est de créer un Stream d'acteurs à partir de la Map précédente, puis de les compter.

À quoi sert la méthode Stream.flatMap() ?

Comment peut-on l'utiliser dans notre cas ?

Pour tester, affichez les 50 premiers acteurs à partir du Stream de tous les acteurs. Utilisez limit pour les 50 premiers et forEach pour l'affichage.

On redéfini la méthode publique toString de *java.lang.String*

|  |
| --- |
| @Override  public String toString**()** **{**  **return** model **+** " " **+** **this.**getYear**();**  **}** |

**3)** Au lieu d'afficher les 50 premiers, comptez le nombre d'acteurs et affichez le résultat.

|  |
| --- |
| /\*\*  \* add a vehicle to rental arrayList  \* **@param** vh  \*/  public void add**(**Vehicle vh**)** **{**  rental**.**add**(**Objects**.**requireNonNull**(**vh**));**  **}** |

**4)** En fait, le calcul précédent est faux car nous pouvons compter le même acteur plusieurs fois, il faut éviter les doublons !

Dans un premier temps, nous allons éviter les doublons en stockant les acteurs dans une structure de données qui a la propriété de ne pas enregistrer les doublons.

Quelle est l'interface Java qui correspond à cette structure de données ?

Quelle implantation de l'interface allons-nous choisir ?

Écrire le code de la méthode numberOfUniqueActors qui prend en paramètre une Map qui associe à un film la liste de ses acteurs et renvoie le nombre total d'acteurs différents ayant joué dans les films de la Map, sous forme d'un entier long.Pour développer la méthode en utilisant un objet StringBuilder et sans retour à la ligne final, on utilise la méthode substring :

|  |
| --- |
| public String toString**()** **{**  StringBuilder st **=** **new** StringBuilder**();**    **for(**Car car**:** **this.**cars**)** **{**  st**.**append**(**car**);**  st**.**append**(**"\n"**);**  **}**  **if(!**cars**.**isEmpty**())** **{**    st**.**substring**(**0**,** st**.**length**()-**1**);**  **return** st**.**toString**();**  **}**  **return** ""**;**  **}** |

**5)** En fait, il existe une méthode Stream.distinct(). Comment peut-on l'utiliser pour trouver le nombre total d'acteurs?

Écrire le code correspondant

Dans

**6)** Rappeler comment marche la méthode collect et les Collector.

Comment peut-on utiliser le collecteur ci-dessus pour grouper les acteurs en fonction d'eux-mêmes ?

Quelle sera le type de retour de l'appel à collect ?

Écrire le code de la méthode numberOfMoviesByActor puis dans le main afficher le nombre de films auxquels a participé Brad Pitt (ou un autre acteur de votre choix), histoire de voir quelque chose !

Pour obtenir un stream à partir d'une liste, on appelle la méthode *stream()* sur l'objet de type

**7)** Il existe une méthode Function.identity(). Comment peut-on l'utiliser dans notre cas ?

Dans le cas ou il n’y a pas de voiture correspondant à l’année demandée, on retourne une ArrayList vide.

**8)** On cherche enfin à écrire une méthode actorInMostMovies qui prend en paramètre la structure de donnée qui associe à un acteur le nombre de film dans lequel il a joué (le résultat de la fonction de la question précédente), et renvoie une paire contenant l'acteur ayant joué dans le plus de film ainsi que le nombre de films dans lequel il a joué.

Sachant que la structure de donnée passée en paramètre pourrait être vide, quel doit être le type de retour de cette méthode?

En Java, il n'existe pas de classe Pair, car il existe l'interface Map.Entry et sa factory method Map.entry(first, second).

En fait, il s'agit de trouver le maximum parmi tous les couples (acteur, nombre de films), et l'on peut utiliser pour cela le collecteur Collectors.maxBy(). sur le stream des couples (acteur, nombre de films) de la structure de donnée passée en paramètre.

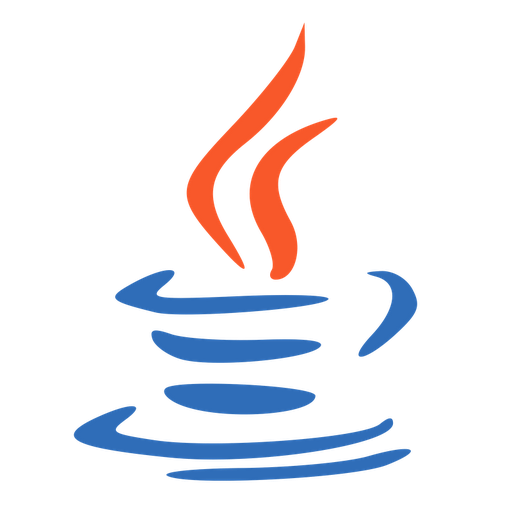
Attention, le comparateur attendu par maxBy ne doit pas être capable de comparer des nombre de films mais des couples (acteur, nombres de film).

On Développe une interface publique *Vehicle*. Aussi, on écrit une classe abstraite *AbstractVehicle* de manière à factoriser le code commun. Le champ yeat est « partagé » entre les classes Car et Camel.

On rédéfini la méthode toString pour avoir l’affichage demandé :

**Conclusion**

Durant ce premier TP de JAVA avancé, j’ai pu revoir des notions importantes de la programmation orienté objet : encapsulation, redéfinition et appel de méthode, exceptions, implémentation d’une interface...



J’ai surtout pu remettre en place les bonnes pratiques (redéfinition *toString, equals & hashcode,* utilisation d’une classe abstraite pour factoriser le code...). Les tests unitaires m’ont permis de me rendre compte si ce que j’avais codé était cohérent avec le résultat attendu.

J’ai pu comprendre l’intérêt de l’API Stream avec des lambda, qui nous demander de penser de façon fonctionnelle et non impérative. En effet, une lambda nous fournit une implémentation pour une interface fonctionnel donnée. Toutefois seul souci, j’ai remarqué qu’il est impossible de lever une exception dans une lambda (à moins de la déclarer dans l’interface fonctionnelle) ce qui m’a un peu embêté au début du TP. J’ai rencontré une difficulté à comprendre comment fonctionne la méthode *map(...)* de l’API stream.

Je dois bien revoir les Exceptions et j’aimerai travailler sur l’implémentation de lambda à l’aide de tests unitaires (avec JUnit 5).