

|  |  |
| --- | --- |
|  | |
|  |  |
| **Projet JAVA :**  **Scooby Doo**  **Buchwald Albane et Vincent Lucile** | Une image contenant texte, clipart  Description générée automatiquement |

**Sommaire :**

[Introduction 3](#_Toc119078010)

[Explication du sujet 4](#_Toc119078011)

[Diagramme de classe 4](#_Toc119078012)

[Détails du code 5](#_Toc119078013)

[Manipulation 5](#_Toc119078014)

[Sentiment 5](#_Toc119078015)

[Personnage 6](#_Toc119078016)

[Suspect 6](#_Toc119078017)

[Coupable 7](#_Toc119078018)

[Innocent 8](#_Toc119078019)

[Animal 9](#_Toc119078020)

[Detective 10](#_Toc119078021)

[DetectiveAnimal 12](#_Toc119078022)

[Histoire 13](#_Toc119078023)

[MyEmptyListException 18](#_Toc119078024)

[EmptyFileException 18](#_Toc119078025)

[Difficultés 19](#_Toc119078026)

[Ce dont nous sommes fières 21](#_Toc119078027)

[Conclusion 22](#_Toc119078028)

# Introduction

Dans le cadre du module de base de Java1, il nous a été demander de réaliser un projet. Ce projet a pour thème de trouver un sujet qui nous plait pour pouvoir confectionner un programme répondant à certains critères. Il peut s’agir de créer un jeu, une histoire ou autre. Cependant, le projet doit comporter toutes les notions de Java que l’on a pu voir en cours.

Après réflexion, nous avons choisi de créer une histoire inspirée d’un des dessins animés de notre enfance : Scooby-Doo.

Dans ce dossier, on va d’abord expliquer notre sujet notamment avec le diagramme de classe. Puis, nous expliquerions différents morceaux du code. Par la suite, nous aborderons les difficultés rencontrées et les solutions. Pour finir, on mentionnera les éléments qui nous rendent fières de ce projet.

# Explication du sujet

Comme dit précédemment dans l’introduction, nous avons articulé notre projet autour d’un épisode de Scooby-Doo : *Quoi d'neuf Scooby-Doo ? vol.2 : Le safari*

Cet épisode se passe dans une réserve naturelle d’Afrique du Sud, où des disparitions étranges d’animaux se produisent. Dans notre histoire, qui se caractérise comme l’épisode, nous suivrons différents personnages :

* Scooby-Doo : chien détective
* Samy : détective
* Vera : détective
* Daphné : détective
* Fred : détective
* Honnie Enseker : touriste et femme de Henri Enseker
* Henri Enseker : touriste et mari de Honnie Enseker
* Jacko : petit singe domestique à chapeau
* Lloyd Emboukou : propriétaire de Jacko
* Capitaine Enrobi
* Docteur Goudefiou : zoologiste

Le but est de découvrir pourquoi les animaux disparaissent et qui se cache derrière tout ça. Pour cela, il faut suivre l’histoire et résoudre les différentes énigmes qui vont nous conduire à la vérité. Attention à faire les bons choix !

## Diagramme de classe

# Détails du code

Passons maintenant à l’explication du code. Comme on peut le voir sur le diagramme, il est composé de plusieurs classes, reliées entre elles ou non.

## Une image contenant texte Description générée automatiquement Manipulation

Cette interface comporte deux méthodes utilisant le même nom. C’est un overload. Ces méthodes servent à afficher une phrase par défaut en fonction de si on l’utilise avec un paramètre de type « String » ou sans.

## Une image contenant texte Description générée automatiquement Sentiment

Cette énumération contient les différents sentiments que les personnages vont ressentir au cours de l’histoire. Le constructeur « Sentiment » sert à assigner le sentiment demander à notre variable. La méthode « toString », définit par Java en override, permet de retourner le sentiment que l’on a assigné.

## Personnage

Une image contenant texte

Description générée automatiquement Le constructeur « Personnage » prend en paramètre le nom et la personnalité de nos différents personnages. Les deux accesseur qui sont à la suite permettent de renvoyer le nom via le « getNom() », et la personnalité via le « getPersonnalite() ».

La méthode « parler » prend en paramètre un String qui est en fait une phrase d’un des personnages. On utilise cette méthode pour pouvoir afficher le nom du personnage et : devant la phrase pour comprendre qui parle. Puis la méthode « sePresenter » va réutiliser « parler » pour cette fois afficher le nom du personnage et les : avec à la suite une phrase par défaut qui va être complété par le *nom* et la *personnalite* du personnage.

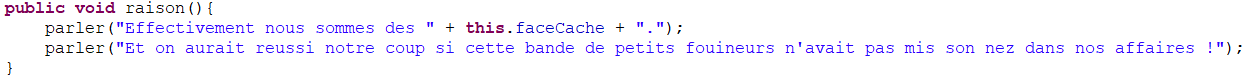
## Une image contenant texte Description générée automatiquement Suspect

Dans ce constructeur, on utilise le constructeur « super() » pour pouvoir récupérer les paramètre de la classe mère « Personnage ». Cela sert à pouvoir les initialiser et les utiliser dans « Suspect ». De plus, on utilise « this.alibi » pour faire la différence entre le paramètre et la variable globale.

Une image contenant texte

Description générée automatiquement L’accesseur « getAlibi() » ne sert qu’à renvoyer l’alibi, qui est utiliser par la suite dans la méthode « excuse ». Cette méthode réutilise la méthode « parler », qui est hérité de « Personnage », et présente un début de phrase par défaut suivi de l’alibi du personnage.

## Une image contenant texte Description générée automatiquement Coupable

 On crée le constructeur de la classe « Coupable ». On réutilise le constructeur « super » de la classe mère « Suspect » et « this ». On utilise également la fonction « sePresenter() » qui vient de l’héritage de « Suspect » qui a elle-même hérité de « Personnage ».

Une image contenant texte

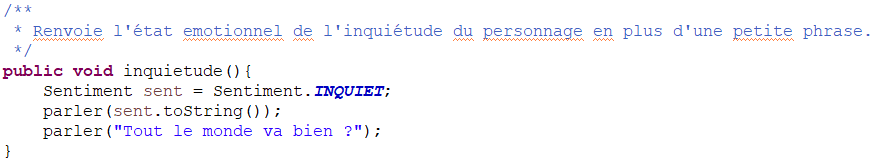
Description générée automatiquementCette méthode réutilise la méthode « parler » avec une autre phrase par défaut et utilise aussi le paramètre « this.faceCache » qui permet aux coupables de donner les raisons de leurs actions.

Une image contenant texte

Description générée automatiquement« fuite » permet de faire éprouver l’émotion déception au coupable qui appelle cette méthode et l’affiche grâce à la méthode « parler ». Puis, grâce à cette même méthode, on affiche une petite phrase pour habiller l’action.

Ici nous avons un overloading car nous avons deux méthodes de même nom « joueFlute ». Leur mission est la même, changer la valeur du booleen « mechant » en true. Cependant, l’une va le faire pour le personnage de type Animal, et l’autre pour le type DetectiveAnimal. Pour cela, Java va comprendre quelle méthode utilisée par rapport aux paramètres qu’on lui donne.

## Une image contenant texte Description générée automatiquement Innocent

**Une image contenant texte

Description générée automatiquement On a la classe « Innocent » qui hérite de « Suspect ». On a également le constructeur qui est créer de la même manière que les classes précédentes.

Une image contenant texte

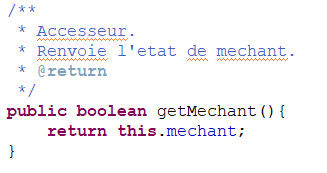
Description générée automatiquementCes méthodes « remerciement » et « inquiétude » agissent de la même façon que « fuite » de la classe « Coupable » en énonçant une phrase par défaut, puis le sentiment *Joie* ou *Inquiet*.

« Nourrir » sert, comme son nom l’indique, à nourrir Scooby en lui donnant des *scoobySnack* définit plus haut dans la classe. On utilise la condition *if* pour voir s’il reste des *scoobySnack* en stock. Si *scoobySnack==0*, le personnage qui appelle cette méthode dit une phrase prédéfinie. Sinon, il en donne un a Scooby et affiche son stock diminué.

## Une image contenant texte Description générée automatiquement Animal

Une image contenant texte

Description générée automatiquement La classe « Animal » hérite de « Personnage » et implémente l’interface « Manipulation ». On a créé comme pour les autres classes le constructeur « Animal ».



Une image contenant texte

Description générée automatiquement Avec le premier accesseur « getMechant », on renvoie l’état de méchant pour pouvoir modifier sa valeur dans la classe actuelle. Et c’est avec la mutateur « setMechant » qu’on va pouvoir modifier sa valeur en lui donnant celle qu’on prend en paramètre.

Cette méthode permet de faire réagir notre animal en fonction de la valeur du booléen *mechant* grâce à la condition *if*. Si *mechant==true*, on affiche le sentiment *Mechant* et on appelle la méthode « attaquer ». Sinon, il dit une phrase par défaut « J’apprends à coder en JAVA !!! ».

## Une image contenant texte Description générée automatiquement Detective

Une image contenant texte

Description générée automatiquement « Detective » est une classe qui hérite de « Personnage ». Cette classe est créer pour définir les détectives tel que Samy, Vera, Daphné ou encore Fred. On définit le constructeur et on utilise un *override* pour redéfinir la méthode « sePresenter » qui est hérité de la classe « Personnage ».

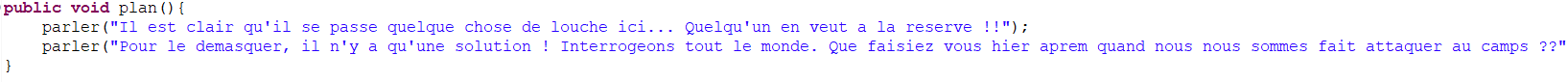
On va gérer dans la méthode « indiceTrouve », les exceptions tel que la lecture d’un fichier vide avec EmptyFileException que nous verrons plus tard. Throws permet la gestion d’exception dans l’instance appelante et ne fait pas de bloc exception. Le try-catch ici va tester tout d’abord de lire le fichier avec un BufferedReader. Si jamais la ligne que l’on lit est nulle (vide) alors on renvoie l’exception du fichier vide : on va chercher le message avec le catch et on affiche “Le papier est vide...” à la place du message d’erreur qui empêche le reste du code de s’exécuter (normalement throw interrompt le programme).

On gère également si le code ne trouve pas le fichier demandé dans le répertoire et affiche alors le message déjà présent dans la documentation Java. Sinon, avec un scanner, on lit le fichier et l’affiche ligne par ligne tant que la prochaine ligne n’est pas vide.

Une image contenant texte

Description générée automatiquementPuis, on utilise le bloc *finally* pour exécuter du code qu’il y ait une erreur ou non. Dans ce bloc, on affiche le sentiment *Réflexion* et la phrase « Regardez Dr, qu’est-ce que vous en pensez ? ».

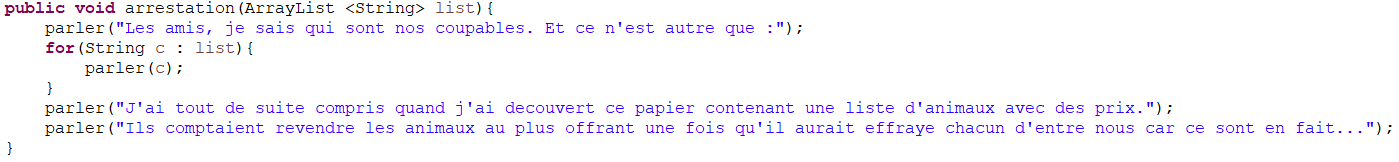
Dans l’histoire, on se retrouve à un moment coincé dans un piège à animaux. C’est à ce moment la que l’on appelle la méthode « piege ». Tout d’abord, on affiche des phrases pour mettre l’utilisateur dans le contexte et on lui affiche une devinette où il faut trouver la date de création de l’ISEN. Pour répondre à ça, l’utilisateur doit écrire le code dans la console. On utilise la boucle *Do…While* pour que, tant que l’utilisateur n’a pas rentrer de code, on utilise le scanner. Ici, le *try* sert à convertir le *String* rentré dans le scanner en un *integer.* Le *catch* sert ici si le *String* rentré dans le scanner n’est pas du format attendu. C’est-à-dire s’il ne peut pas être converti en *integer*.

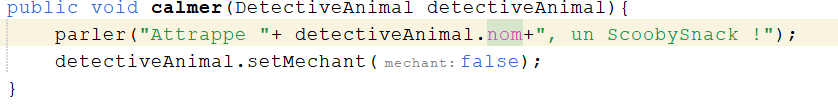
 À la suite de cette boucle, on utilise la condition *if* pour vérifier si l’utilisateur a rentré le bon code ou non. Si c’est le cas, on affiche le sentiment *Joie* et une autre phrase. Sinon, on affiche une autre phrase pour dire à l’utilisateur que ce n’est pas le code attendu et on rappelle la fonction « piege » pour reprendre du début.

Une image contenant texte

Description générée automatiquement « plan » ne sert qu’à afficher des phrases pour agrémenter l’histoire.

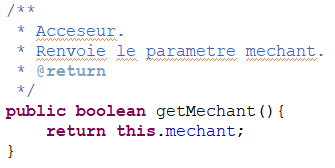
Cette méthode « poursuite » agit de la même façon que « fuite » de la classe « Coupable » en énonçant une phrase par défaut mais cette fois avec un nom au milieu de celle-ci, puis le sentiment *Peur*.

« arrestation » permet de dévoiler qui sont les coupables en prenant leurs noms en paramètres dans une liste de String et en donnant des explications pour faire sens à l'histoire. On utilise la boucle *for* pour afficher le nom pour chaque String de *list*.

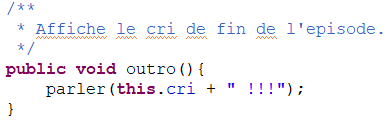


Cette méthode sert à pouvoir modifier la valeur du booléen *mechant* de l’objet qui est passé en paramètre en lui attribuant la valeur *false*. Puis, on affiche également une petite phrase toujours pour continuer le scénario de l’histoire.

## Une image contenant texte Description générée automatiquement DetectiveAnimal

Une image contenant texte

Description générée automatiquementLa classe « DetectiveAnimal » hérite de « Detective » et implémente l’interface « Manipulation ». On a créé comme pour les autres classes le constructeur « DetectiveAnimal ».

Comme pour la classe « Animal », on retrouve les accesseurs « getMechant » et « setMechant », qui renvoie l’état de méchant pour pouvoir modifier sa valeur dans la classe actuelle en lui donnant celle qu’on prend en paramètre.

« outro » ne sert qu’à afficher le cri de scooby à la fin de l’épisode pour le conclure.

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement Cette méthode fonctionne comme « fuite » de la classe coupable en utilisant cette fois le sentiment *Faim.*

Cette méthode permet de faire réagir notre *DetectiveAnimal* en fonction de la valeur du booléen *mechant* grâce à la condition *if*. Si *mechant==true*, on affiche le sentiment *Mechant* et on appelle la méthode « attaquer ». Sinon, il dit une phrase par défaut « Miam ! », qui fait sens quand on lit l’histoire complète.

## Histoire

Une image contenant texte

Description générée automatiquement« Histoire » est une classe à part qui n’hérite d’aucune autre classe. Elle permet d’agrémenter l’histoire et de faire participer l’utilisateur activement à la recherche de coupable.

Elle prend en paramètre une *ArrayList de String* (les noms des potentiels coupables) et une deuxième avec les vrais coupables dedans. Elle va être remplie au fur et à mesure par l’utilisateur s’il le souhaite. Le constructeur ici est vide car il ne nous est d’aucune utilité.

Ci-dessous les accesseurs et mutateur pour modifier la valeur ou plutôt le contenu de nos listes ainsi que d’y accéder.

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquementLa méthode « synopsis » va aller lire un texte Histoire qui initie, introduit notre histoire : ici, l’épisode du Safari.

On va gérer dans cette méthode les exceptions tel que la lecture d’un fichier vide avec EmptyFileException que nous verrons plus tard. Throws permet la gestion d’exception dans l’instance appelante et ne fait pas de bloc d’exception. Le try-catch ici va tester tout d’abord de lire le fichier avec un BufferedReader. Si jamais la ligne que l’on lit est nulle (vide) alors on renvoie l’exception du fichier vide : on va chercher le message avec le catch et on affiche “L’histoire charge...” à la place du message d’erreur qui empêche le reste du code de s’exécuter. On gère également si le code ne trouve pas le fichier demandé dans le répertoire et affiche alors le message déjà présent dans la documentation Java. Sinon, avec un scanner, on lit le fichier et l’affiche ligne par ligne tant que la prochaine ligne n’est pas vide.

Une image contenant texte

Description générée automatiquementOn initie notre objet « Histoire » et on appelle le synopsis.

Le résultat est le suivant :

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementLa méthode suivante « listeSuspect » prend en paramètre une « LinkedList » de String avec le nom de tous les suspects. Cette liste est créée dans la main :

Une image contenant texte

Description générée automatiquementOn va gérer dans cette méthode aussi les exceptions tel que NumberFormatException et MyEmptyFileException que nous verrons plus tard également.

On fait donc un try-catch :

Si jamais la liste de suspect est vide, l’utilisateur n’a plus de pouvoir sur la suite de l’histoire. Il a perdu toutes ses chances de deviner qui sont les coupables. On affiche alors le message suivant « La liste de suspect est vide… Vous ne pouvez plus rien faire. » grâce au catch (MyEmptyListException f) présent plus loin. Sinon on affiche la liste de suspect et on demande à l’utilisateur ce qu’il veut faire : 1 il enlève un suspect de la liste, 2 il met un suspect dans la liste de coupable, 0 on repose la question, le reste correspond à ne rien faire. On vérifie que la réponse rentrée dans le scanner est bien un entier avec le « Integer.parseInt » sinon on renvoie l’exception NumberFormatExecption qui repose la question du début.

3 cas s’offrent donc à la suite.

Le 1e cas permet de supprimer un suspect de la liste selon son index commençant à 1.

Une image contenant texte

Description générée automatiquementOn reprend le même try-catch de tout à l’heure pour vérifier que l’entrée du scanner est un entier.

On vérifie que l’entrée est bien un index possible dans la liste sinon on renvoie à l’erreur NumberFormatException et on repose la question. On enlève 1 à l’entrée pour que l’index commence bien à 0 et on le remove (supprime) de notre liste de suspect.

On affiche à la fin notre liste de suspect modifiée.

Une image contenant texte

Description générée automatiquementLe 2e cas permet de passer un suspect dans la liste des coupables selon son index commençant lui aussi à 1.

Même principe que précédemment mais ici on ajoute le suspect dans la liste de coupable de notre classe avec le « getList.add(liste.get(y-1)) ». Le « .get(y-1) » va chercher le nom à l’index y-1 et l’ajoute « .add » à notre liste.

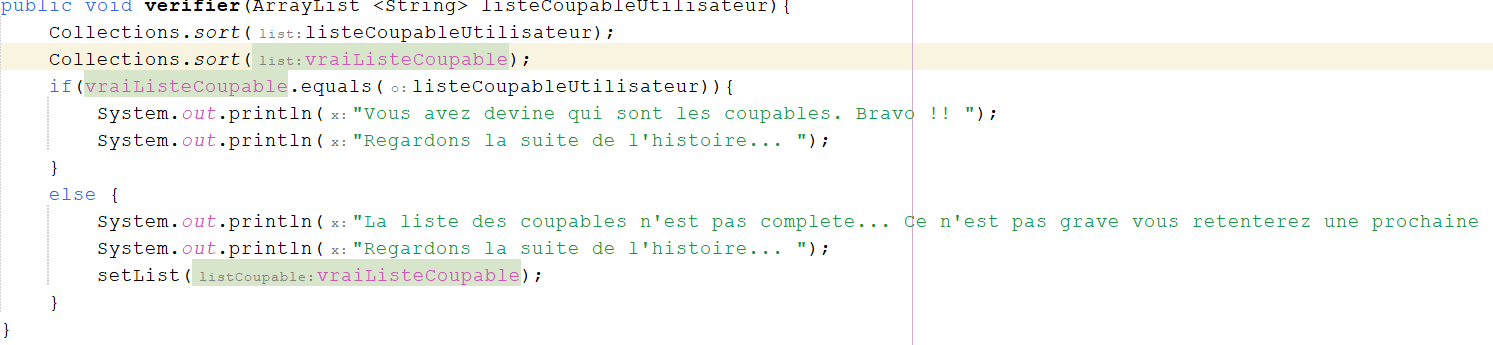
Le « continue » permet de continuer le code malgré l’exception.

De plus on le supprime de notre liste de suspect avec le même code que précédemment.

Une image contenant texte

Description générée automatiquementLes autres cas sont regroupés ensemble :

On ne fait rien et notre histoire continue son cours. On retrouve bien à la fin du try, le catch(MyEmptyListException) évoquée précédemment qui va chercher le message.

Une autre méthode importante de notre classe est « vérifier ».

Cette méthode prend en paramètre la liste de coupable que l’utilisateur a modifié (ou non) et va la comparer à une « ArrayList » de String contenant les coupables à trouver.

On trie nos deux listes avec « .sort » pour pouvoir par la suite comparer leurs contenues quelque soit l’ordre des coupables rentrés dans la liste.

Selon si les listes sont égales ou non, on affiche des paragraphes de texte différents. Si jamais la liste de coupables faite par l’utilisateur n’est pas complète ou fausse, on la modifie pour qu’elle soit égale à la vraie.

On voit si dessous la méthode pour remplir la vraie liste de coupable.

Une image contenant texte

Description générée automatiquement



Une image contenant texte

Description générée automatiquementAinsi que les lignes dans le main pour remplir la liste. Elle va donc dépendre les noms donnés aux objets « Coupable ».

Dernièrement, la méthode « remerciementDeFin » pour remercier l’utilisateur d’avoir participer et affiche les noms prénoms des développeurs stockés dans un tableau.

## Une image contenant texte Description générée automatiquement MyEmptyListException

La classe « MyEmptyListException » est une exception créée et appelée dans le cas de liste vide. Elle hérite de la documentation de la classe « Exception » présente dans Java.

Cette dernière renvoie des messages personnalisés avec l’accesseur pour l’affichage « getMessage ». On l’utilise comme vue précédemment dans la classe « Histoire », précisément dans la méthode « listeSuspect ».

## Une image contenant texte Description générée automatiquement EmptyFileException

La classe « EmptyFileException » est une exception créée et appelée dans le cas de liste vide. Elle hérite de la documentation de la classe « Exception » présente dans Java. Cette dernière renvoie des messages personnalisés avec l’accesseur pour l’affichage « getMessage ».

On l’utilise comme vue précédemment dans la classe « Histoire », précisément dans la méthode « synopsis » mais également dans la classe « Detective » dans la méthode « indiceTrouve ».

Créer sa propre exception est très utile pour la gestion d’exception selon ce qu’on veut faire d’une exception/erreur. Ici nous évitons que tout le code s’arrête en ne renvoyant que des messages souhaités.

# Difficultés

Au cours du projet, différentes difficultés se sont présentées à nous.

Tout d’abord, pouvoir accéder au paramètre *méchant* de la classe « Animal » et « DetectiveAnimal » pour pouvoir le modifier dans la classe « Coupable » si la méthode « joueFlute » est appelée. Nous avions au début pensé à créer des singletons de classe avant de se rendre compte que c’était bien trop difficile à mettre en place.

Une image contenant texte

Description générée automatiquementPar exemple dans la classe « Animal » nous avions créé une instance de la classe :

Une image contenant texte

Description générée automatiquementLe constructeur permettait lors de la création de l’objet « Animal » de le mettre dans le paramètre *instance.*

La méthode du « getInstance » permettait d’accéder à ce paramètre et de pouvoir donc accéder au setMechant et de le modifier en le passant à *true* dans la méthode joueFlute() de la classe « Coupable ».

Néanmoins nous avons vite compris que nous ne voulions pas de singletons et qu’il aurait été difficile d’empêcher la création de plusieurs objet « Animal » et « DetectiveAnimal ».

Nous avons donc passé en paramètre de la méthode « joueFlute » l’objet Animal et nous avons recréé la même méthode en dessous avec cette fois si l’objet DetectiveAnimal en paramètre donc un overload.

Deuxième difficulté rencontrée était la gestion de fichier vide pour le « synopsis » et « indiceTrouve ». Aucune exception de ce type n’existait sur Java ce qui nous a conduit à créer notre propre exception. De même pour les listes vides dans notre cas si jamais vous vouliez afficher la liste des suspects ou la modifier (listCoupable).

Nous avons donc créé deux classes qui hérite de « Exception ». Elles ont donc accès au message d’erreur de cette classe par l’appel du constructeur super. On peut donc personnaliser notre message si notre liste est vide et gérer l’exception sans que cela affecte le code. La compréhension de comment utiliser « throw new » et « catch(MyEmptyListException e) » nous a pris du temps. Nous créions des boucles ou des conditions mais pas de réelle exception pour traiter le problème.

Également s’en est suivit la recherche pour ne pas afficher le message d’erreur de Java mais simplement notre message. A chaque exception rencontrée, le message d’erreur de Java s’affichait ce qui stoppait le code. Si jamais l’exception était rencontrée, nous voulions que notre code l’ignore ou continue simplement le code. D’où l’utilisation du continue pour ignorer l’exception et continuer le code. L’affichage du message avec par exemple le throw new EmptyFileException(« … ») par le catch affiche notre message sans l’exception de Java.

L’utilisation de LinkedList nous a également poser un problème. Nous ne savions pas où les utiliser et comment. Nous sommes parties sur l’idée au début de faire une liste de suspect qui s’initierait toute seule à la suite de la création des « Coupables » et « Innocents » dans la classe « Suspects. Malheureusement nous n’arrivions qu’à ajouter le nom du dernier « Suspect » créé. Nous avons ensuite testé de faire deux LinkedList séparée, une dans la classe « Coupable » et une dans la classe « Innocent » que nous aurions fusionnée dans la classe « Suspect ». Le même problème s’est posé : seul le dernier « Coupable » et « Innocent » s’ajoutait. Nous avons donc fini par l’initialiser dans le *main* en ajoutant manuellement par code dur les « Suspect » et l’appeler dans la méthode « listeSuspect » dans la classe « Histoire ». C’est d’ailleurs cette liste qui a entrainé la création de la classe « Histoire » qui agrémente désormais l’histoire et permet plus d’interactivité avec l’utilisateur qui peut jouer avec cette LinkedList.

Pour les scanners nous voulions que l’utilisateur rentre un chiffre compris entre 1 et 9 et si quelque chose d’autre été appelé nous l’ignorions ou recommencions le scanner jusqu’à ce que le format soit correcte.

Une image contenant texte

Description générée automatiquementNous avons donc dû réaliser une boucle :

L’entrée du scanner est un String. Nous le transformons avec un « Interger.parseInt() » en entier que nous stockons dans une variable x.

Si x=0, nous recommençons le scanner.

Si x n’est pas un entier, l’exception « NumberFormatException » est interceptée. On prévient que le format n’est pas correct et nous reprenons notre scanner puisque x est toujours égale à 0. Il n’a pas pu changer de valeur.

# Ce dont nous sommes fières

Apprendre le Java à travers des cours ne permet pas d’apprendre à le coder. Ce projet est donc une bonne application de ces cours. Nous sommes donc contentes d’avoir réussie à créer un projet complet, utilisant tout ce que l’on a vu pendant ce module.

De plus, le projet étant totalement libre, il est difficile de trouver une idée qui nous permettrait de réunir tout ce que nous avons appris. Nous sommes donc satisfaites de notre sujet qui se porte sur Scooby-Doo, car c’est un dessin animé de notre enfance qui nous a marqué. Nous avons donc repris le thème d’un épisode en le réadaptant pour qu’il rentre dans les critères de ce projet.

La contrainte de temps était aussi un défi de taille car nous nous sommes fixées comme objectif de rendre ce projet 1 semaine à l’avance afin d’obtenir le point bonus, juste comme défi personnel. Mais aussi car, en plus de ce projet ci, nous avons chacune de notre côté d’autres projets ainsi que les partiels qui arrivent à grand pas.

Puis, comme dans tout projet, rien ne se passe comme prévu et rien ne fonctionne du premier coup. La persévérance dont nous avons fait preuve pour ne pas abandonner face aux difficultés rencontrées et le fait de les avoir surpassées nous rendent fières. Car nous avons réussie à travailler en équipe pour résoudre ces difficultés et à prendre le temps de chercher et comprendre le langage Java.

Nous avons aussi ajouté nos touches d’humour car nous voulions prendre plaisir à lire et créer cette histoire.

Pour finir, nous sommes bien entendu très contentes de rendre un projet concret et proprement terminer, c’est à dire pas dans la précipitation. C’est le détail qui nous à fait le plus plaisir dans le rendu de ce projet, même si nous sommes conscientes que ce n’était pas le sujet le plus compliqué que ce projet peut proposer.

# Conclusion

En conclusion, rétrospectivement, nous sommes satisfaites du rendu de ce projet car nous avons atteint tous les objectifs que nous avions fixés au début de celui-ci. Ce projet nous a permis de comprendre et de mettre réellement en application ce que nous avons appris lors des séances de cours de ce module.

De même, le fait de pouvoir choisir notre sujet de A à Z nous a vraiment plu car nous avons pu créer quelque que chose qui plait et qui nous a un peu plus motivé que si c’était un sujet imposé. Nous sommes d’ailleurs convaincues que nous avons pris plus de plaisir que si nous avions tenté de trouver un sujet apportant plus de difficultés. Compte tenu de notre niveau, nous avons fait le choix de troquer la difficulté à la maitrise. Mieux vaut un projet qui fonctionne et qui est propre qu’un brouillon incompréhensible qui ne nous correspond pas.

Nous sommes heureuses de clôturer ce module sur cette touche d’amusement en duo.