Le mystère des ruines Sheikah

A3P Java 2025/2026 G2

Description du jeu

Auteur

Benoît de Keyn

Thème

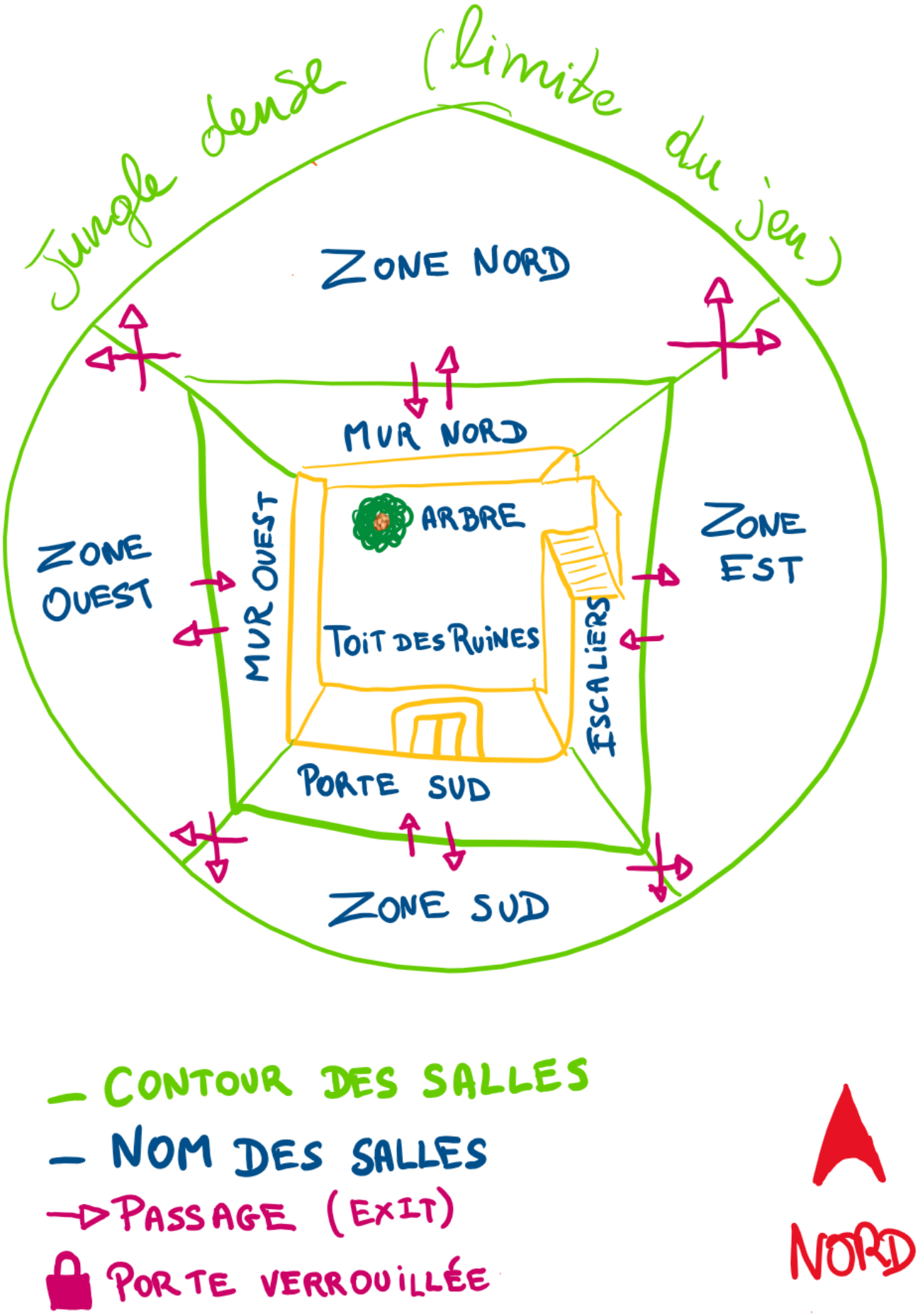
Dans des ruines anciennes, un archéologue doit trouver un artefact du peuple Sheikah.

Résumé du scénario

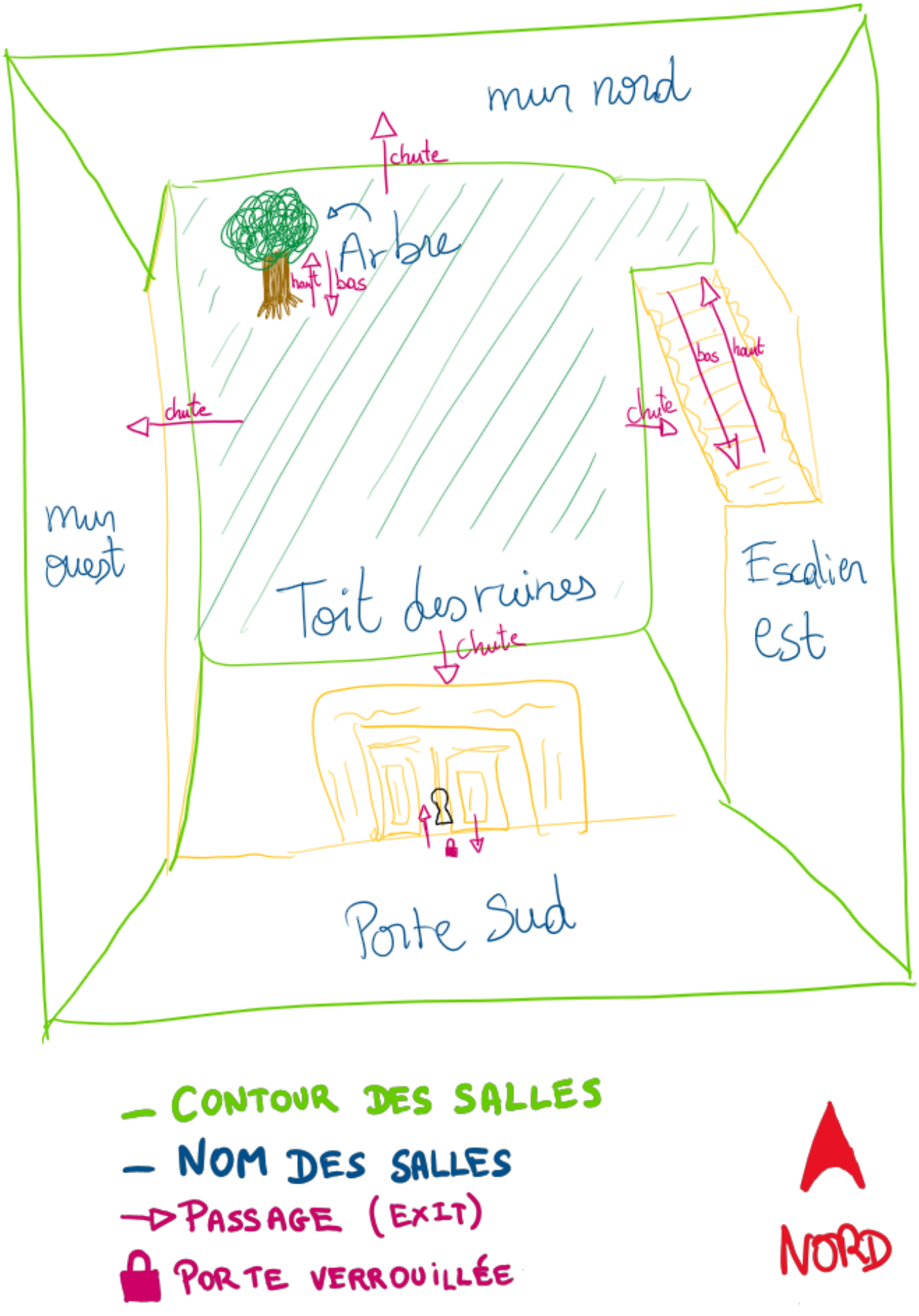
Un archéologue Sheikah découvre un manuscrit vieux de 5000 ans mentionnant un artefact capable de transformer la chaleur en mouvement. Il part à sa recherche dans la jungle Korogu, où une ruine mystérieuse l'attend.

Plan

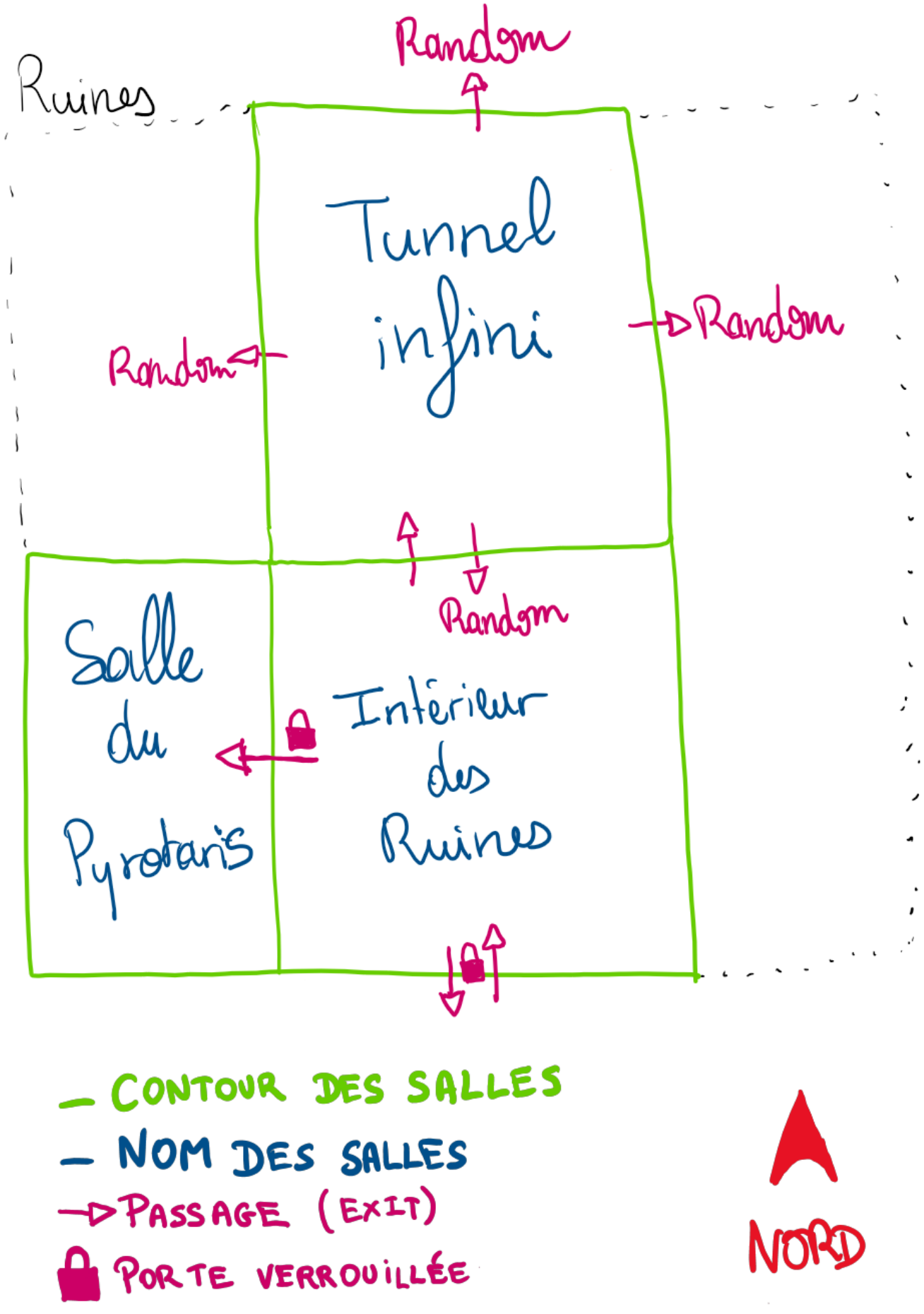
Vue d’ensemble aérienne

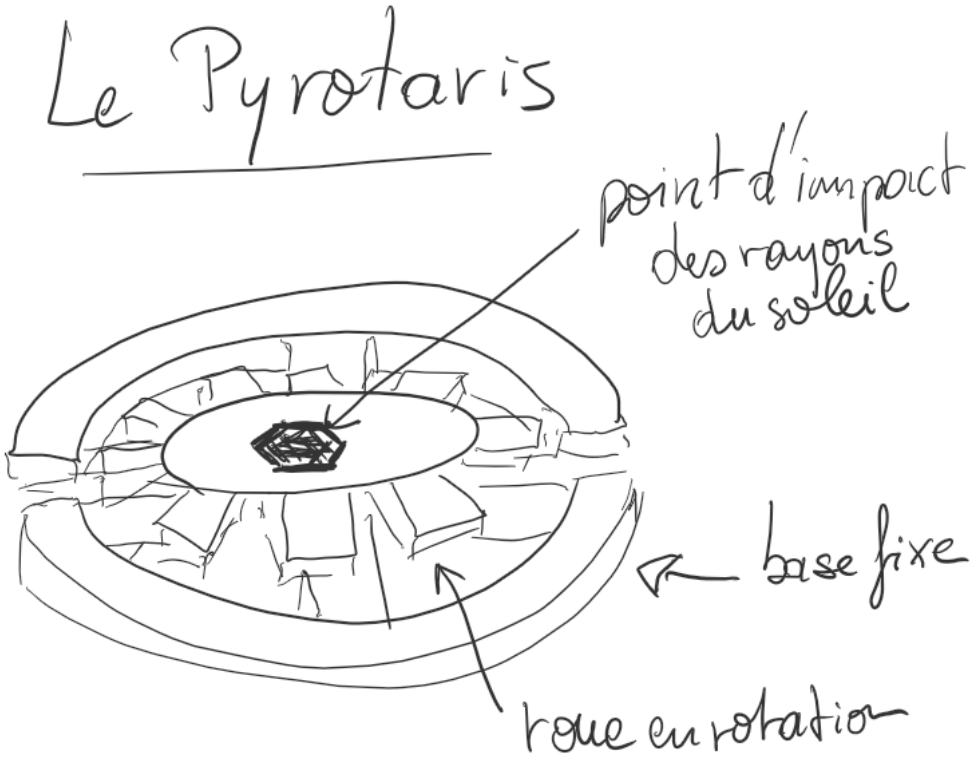


Vue 3D extérieure des ruines



Intérieur des ruines





Scénario détaillé

Un archéologue expert dans la civilisation Sheikah parvient à déchiffrer un vieux manuscrit datant de 5000 ans qui mentionne une pièce exceptionnelle de technologie qui permettrait de convertir la chaleur en mouvement. Cela expliquerait comment ils sont parvenus à réaliser des monuments titanesques il y a 5000 ans.

Il décide de se rendre dans ce qui semble être la jungle Korogu pour découvrir cet artefact.

Arrivé devant une ruine souterraine Sheikah au milieu de la forêt équatoriale, il doit trouver l'artefact. L'archéologue commence son aventure dans la zone sud, devant les ruines.

Détail des lieux, items, personnages (non exhaustif)

- Le pyrotaris : trésor à récupérer
- Le rocher pour ouvrir la porte de la salle du pyrotaris
- Le tunnel infini qui vous envoie aléatoirement dans une salle du jeu

Situations gagnantes et perdantes

- Situation stagnante :
  L'archéologue cherche le pyrotaris mais il lui reste encore des actions disponibles.
- Situation perdante :
  L'archéologue a atteint le nombre maximum de déplacement sans avoir trouver le pyrotaris.
- Situation gagnante :
  L'archéologue trouve et prend le pyrotaris, dans le nombre de déplacements imparti.

Énigmes, mini-jeux, combats

- Aller chercher la clé dans le nid pour ouvrir la porte des ruines.

- Ingérer la fiole de peroxyde d'hydrogène pour avoir la force de prendre le rocher dans son inventaire.
- Déverrouiller la salle finale avec le rocher comme levier.
- Le but final est de récupérer le Pyrotaris (pas seulement le trouver).

Comment Gagner

- Se déplacer jusqu'aux escaliers.
- Monter 2 fois d'affilée pour aller dans l'arbre.
- Prendre la clé.
- Se déplacer jusqu'au mur Nord des ruines.
- Prendre la fiole.
- Ingérer la fiole pour avoir la force de prendre le rocher.
- Se déplacer jusqu'au mur Ouest.
- Prendre le rocher.
- Se déplacer jusqu'à la porte Sud.
- Déverrouiller la porte Sud (grâce à la clé).
- Aller à l'intérieur des ruines.
- Déverrouiller la porte Ouest (grâce au rocher).
- Aller dans la salle finale derrière cette porte.
- Prendre le Pyrotaris.

Réponses aux exercices

Exercice 7.3.3

J'ai déplacé le scénario réduit en tant que scénario final après l'avoir étoffé un peu.

Exercice 7.4

J'ai pris la version réduite du scénario pour implémenter mes rooms.
J'ai créé autant de variables de type Room que de zone de mon croquis, puis j'ai assigné les sorties nord, est, sud, est selon la direction du passage et le sens des pièces.

Aussi, j'ai traduit les commandes quit, help et go, ainsi que les textes affichés. J'ai adapté les textes d'introduction et de la commande 'aide' à mon jeu. Aussi j'ai personnalisé un peu les messages d'erreur pour plus de clarté.

Exercice 7.5

J'avais déjà empêché cette duplication de code lors du TP, mais uniquement pour afficher les sorties.
J'ai donc renommé la fonction, et ajouté la description de la room actuelle.
J'ai personnalisé un peu la fonction pour un affichage plus clair avec des flèches et des retours à la ligne.
J'ai aussi changé les descriptions de mes lieux pour que la phrase « Vous êtes … » soit cohérente avec la suite.

Exercice 7.6

J'ai mis en privé tous les attributs de la classe Room, puis créé une fonction accesseur getExit() qui prend en paramètre la chaîne de caractère qui correspond à la direction pDirection.

Au lieu de faire une suite de « if else » avec des equals, j'ai préféré un switch case sur le paramètre pDirection (car le switch case prend en charge les égalités de Strings nativement). Si la direction demandée n'est aucun des 4 points cardinaux, elle retourne la référence null.

Dans la classe game, dans goRoom, j'ai aussi préféré un switch case sur le 2nd mot de la commande (à savoir la direction).
Si le second mot ne correspond à aucun point cardinal, il y a le message d'erreur « Direction inconnue ! » Autrement, la variable vNextRoom stocke la référence renvoyée par getExit().

Ensuite, on vérifie si cette référence est null. Le cas échéant, on affiche le message d'erreur « Vous ne pouvez pas aller dans cette direction ! »

J'ai peut-être omis une situation d'erreur, mais je n'ai pas l'impression d'avoir eu besoin d'une des 3 solutions proposée pour prendre en charge une mauvaise direction demandée…

Exercice 7.7

Pour cet exercice j'ai créé la fonction getExitString() qui créée une variable contenant le début du message : « Les directions possibles sont : » .
Puis, avec une suite de 4 conditions indépendantes pour chacun des 4 points cardinaux, elle concatène (ou non) la direction formattée dans un format plus clair à lire pour le joueur.
La String est ensuite retournée.

Cette fonction est appelée dans printLocationInfo() juste après la description.

Il est préférable que les sorties disponibles soient lues dans la classe Room pour mieux garder les fonctions qui utilisent les attributs d'un objet Room uniquement dans la classe Room.

Exercice 7.8

Dans la classe Room j'ai :

- Ajouté un attribut exits, une hashmap
- Ajouté la ligne du constructeur pour initialiser exits.
- Changé setExit() comme dans le livre, avec exits.put(pDirection, pNeighbor)
- Changé getExit() en renvoyant exits.get(pDirection) après avoir vérifié si la clé existe, sinon, null.
- Changé getExitString() en utilisant un for-each qui itère sur la liste des directions exits.keySet()

En revanche je n'ai pas eu besoin de la ligne import java.util.set comme proposé dans l'aide (j'ai peut-être pris une version trop récente de BlueJ, ou la version de Java inclue 'set' dans 'util')

Dans la classe Game j'ai :

- Changé dans la méthode goRoom, l'assignation de vNextRoom en la réduisant à une simple ligne qui appelle getExit(pDirection)
- Modifié l'appel de setExit() pour chaque connexion entre 2 pièces pour l'adapter à la nouvelle méthode, comme dans le livre.

Maintenant, on ne peut effectivement plus faire la différence entre une direction bloquée ou une direction non reconnue, car tous les noms de direction peuvent être ajouté en tant que clé dans la hashmap.

Pour résoudre ce problème, j'ai créé une liste des 6 directions possibles dans mon jeu, et goRoom vérifie si la direction demandée appartient à ce set pour spécifier le message d'erreur.
J'ai repris ce qui est fait dans la classe commandWords. J'y ai rajouté la liste des 6 directions pour vérifier si la direction souhaitée existe, avant d'aller appeler aCurrentRoom.getExit dans goRoom.

Pour cela j'ai modifié les dernières lignes de getCommand() de la classe Parser, en rajoutant des tests sur vWord2 (si elle vaut null et si aValidCommands.isDirection(vWord2) est vrai) en fonction, le second word pour la commande go est soit : une direction valide, soit "invalid", soit null). Ces 3 valeurs sont vérifiées dans goRoom pour afficher le bon message d'erreur.

Exercice 7.8.1

J'ai ajouté un arbre sur le toit des ruines, avec les 2 directions « haut» et « bas» entre l'arbre et le toit.
Aussi j'ai remplacé les directions est/ouest pour passer de l'escalier au toit par bas/haut.

Exercice 7.9

Ah mince, j'avais déjà utilisé une boucle for each pour getExitString. J'utilisais déjà ce type de boucle l'année dernière, surtout en javascript avec for (value of array) { … }.
Et la concaténation de String avec += aussi m'a paru naturelle.

Ma fonction n'a donc pas changé ici, par contre elle est différente du livre dans le sens où je n'ai pas de variable intermédiaire qui stocke la liste des directions possible mais directement :
for (String vDirection : this.exits.keySet()) { … }
Mais je comprends bien la différence entre un set (Set<Type>) et un tableau Type[], et que la méthode keySet() renvoie un set et non un tableau.

Exercice 7.10

On commence par créer une chaîne de caractères initiale contenant le texte d'introduction : « Les directions possibles sont : ».
Cette String est la base pour construire le message final.
La méthode ne reçoit aucun paramètre.

Sur la même ligne, on déclare un Set (collection d'éléments uniques) de type String et on lui assigne le résultat de la méthode .keySet() appliquée à la hashmap aExits qui contient les différentes associations { clé String -> référence d'une Room } qui correspondent aux différentes directions possibles (les clés) associées aux différentes Rooms de sorties.

Une boucle for each répète le code entre crochets autant de fois qu'il y a de clés (donc de directions) trouvées dans la hashmap aExits de la room désignée par this.
A chaque fois que la ligne entre crochet s'exécute, la variable vDirection prend pour valeur une des clés de a hashmap.

La ligne ainsi répétée concatène à la string de base cette direction pour pouvoir finalement retourner à la fin de la méthode, toutes les directions possibles .

Exercice 7.10.1

J'ai écrit plein de commentaires javadoc pour expliquer le but de chaque méthode, ses paramètres et ses sorties. Notamment avec les mots clé comme @param et @return
J'ai traduit en français aussi les commentaires de Parser, Command et Command Words pour la cohérence.
Je me suis ajouté en co-auteur pour les classes Parser et Command
J'ai changé le nom du jeu.

Exercice 7.10.2

J'ai résolu des erreurs de génération dans BlueJ en allant dans chaque classe puis en affichant la vue Interface. Alors les classe devenaient hachurée pour être recompilée. Cela corrigeait des erreurs que je ne connais toujours pas, mais j'ai l'impression que ça n'a rien changé à mes commentaires.

Bien sûr, la classe game est comme une fonction main, qui sert à avoir une game loop et à centraliser les appels des différentes fonctions définies ailleurs, afin de bien découpler les classes, et de pouvoir modifier du code sans avoir à changer des appels sur toutes les couches.

Exercice 7.11

J'ai ajouté à la classe Room la méthode getLongDescription qui renvoie la String que printLocationInfo affichait, et j'ai réduis cette dernière dans game, à simplement afficher le résultat de cette nouvelle méthode.

Exercice 7.14

J'ai renommé printLocationInfo par look dans la classe Game et renommé les 2 appels à cette fonction.
J'y ai aussi ajouté une clause dans le switch case de processCommand qui renvoie vers look si la commande est « regarder »

Dans CommandWords, j'ai eu une erreur parce que j'ai rajouté un String « regarder » au tableau des commandes valides, car il était encore écrit comme dans le TP.
J'ai donc changé la déclaration des attributs conctant en leur assignant directement la liste de valeurs comme ils font dans le livre, et aussi supprimé le contenu du constructeur de CommandWords.

Exercice 7.15

J'ai rajouté la méthode breathe() qui affiche "Vous venez de consommer une bouffée d'oxygène de votre réserve."
J'ai ajouté la String « respirer » à la liste des commandes, et comme pour « regarder », une clause au switch case de processCommand.

Exercice 7.16

J'ai ajouté

- La méthode showAll à CommandWords qui un peu de la même manière que pour getExitString, utilise une boucle for each mais cette fois sur un tableau de Strings, pour afficher la liste des commandes valides.
- La méthode showCommands à Parser qui se contente d'appeler showAll.
- L'appel à showAll dans la méthode printHelp de la classe Game.

Exercice 7.17

Oui, il faut changer Game à chaque fois qu'on ajoute une nouvelle commande valide.
Car maintenant que cette commande est disponible, il faut lui assigner une action à exécuter quand le joueur l'utilise dans processCommand.

Exercice 7.18

J'ai échangé la méthode showAll de CommandsWord par par une méthode dont la signature est :

```
public String getValidCommandsString()
```

et qui renvoie la String qui contient la liste des commandes :
Elle crée une String vide de base, puis avec une boucle for each qui itère sur le tableau des commandes valides, elle concatène chaque commande valide à cette base avec +=, avant de retourner la String ainsi complétée.

Puis dans parser j'ai échangé showCommands par getCommandsList() qui transmet simplement le résultat précédent par un appel de la méthode getValidCommandsString sur l'objet CommandWords.

Finalement dans print help, j'ai mis la ligne :

```
s.o.p(aParser.getCommandsList())
```

au moment d'afficher la liste des commandes.

Exercice 7.18.1

La version zuul-better téléchargée ne compilais pas.
J'ai corrigé la mise en commentaires des lignes de la classe Game qui concernaient les images.
Mais après ça j'ai toujours une erreur de compilation dans Game à l'appel de parser.getCommand(), car dans la classe parser, cette méthode attendait le paramètre pInputLine, alors j'ai modifié cette classe en ajoutant l'import de la bibliothèque Scanner, et j'ai copié ce que nous avions fait dans notre jeu pour que getCommand() ne lise plus le paramètre mais une variable locale vInputLine correspondant à la dernière ligne entrée dans l'invite de commande.
Ensuite j'ai aussi corrigé l'erreur de compilation dans Game() car printHelp appelait showCommands alors que maintenant le parser avait une méthode qui retournait la String contenant la liste des commandes.

Je crois que c'était inutile que zuul-better s'exécute bien parce que ça ne m'a pas empêché de comparer et de voir les différences suivantes :

- Dans la classe CommandWords : outre le fait que j'ai appelée ma fonction qui liste les commandes getValidCommandsString, la première différence est l'utilisation d'une boucle for au lieu de for-each (ce que vous avez dit ne pas changer dans la FAQ). La deuxième différence c'est que j'ai déclaré/assigné un String vide, que j'ai concaténé dans la boucle for-each avec l'opérateur +=, tandis que dans zuul-better, ils utilisent respectivement stringbuilder et .append().
- Dans zuul-better, s'il n'y a pas de sorties dans une direction demandée, la command getExit renvoie null et le message d'erreur est toujours « there is no door ». Tandis que j'ai différencié l'absence de sortie et l'invalidité d'une direction dans mes messages d'erreur en ajoutant une liste des directions valides dans CommandWords et les méthodes correspondantes.

Autrement j'ai lu dans la FAQ que vous préferiez avoir dans la classe Game :

- Une fonction printLocationInfo qui appelle getLongDescription sur aCurrentRoom
- Une fonction look qui se content d'appeler printLocationInfo

Alors que j'avais remplacé printLocationInfo et ses appels par l'unique fonction look().
J'ai donc suivi ce que vous indiquez dans la FAQ en appelant dans Game : soit look() (dans processCommand) soit printLocationInfo (dans printWelcome et goRoom) pour afficher la description de la pièce et les directions disponibles.

Autrement, les autres différences me paraissent négligeables.

Exercice 7.18.2

L'avantage d'utiliser StringBuilder au lieu de simplement manipuler des String, se fait ressentir lorsqu'il faut beaucoup éditer le contenu d'une chaîne de caractères.
En fait si j'ai bien compris, quand on additionne 2 strings, cela créée une nouvelle String qui vaut la somme des 2 précédente, mais gardes les 3 en mémoire. Il y a autant de nouveaux objets de type String qui sont créé à chaque itération de la boucle for-each dans getExitString ou getValidCommandsString. Le type String est dit immutable.
Alors qu'en utilisant Stringbuilder, la variable qui contient la chaîne de caractère devient mutable. Donc on économise de la mémoire, et on garde la même référence au passage.

J'ai donc changé getExitString et getValidCommandsString pour qu'elle utilise StringBuilder comme on peut le voir dans zuul-better : Initialisation (vide ou non), puis des .append(...) qui remplacent += ..., et finalement l'utilisation de .toString() pour générer un objet de type String.

Exercice 7.18.3

10 images créées avec PowerPoint en .png 800x600.
Je les aies mises dans un dossier « images » dans le dossier BlueJ du projet.

Exercice 7.18.4

Le titre du jeu : *Le mystère des ruines Sheikah*

Exercice 7.18.6

En suivant l'énumération au point 2 :

a) Dans **Command** et **CommandWord** : Pas de changement

b) Dans **Room** : j'ai ajouté l'attribut « aImageName », la définition de cet attribut dans le constructeur ainsi que la méthode getImageName().

c) Dans **Parser** :
Au lieu de lire la ligne du terminal dans la fonction getCommand(), on la prend en paramètre. Nous n'avons donc plus besoin du Scanner ni des lignes qui servaient à acquérir ce qui est maintenant donné en paramètre : la ligne du terminal.
Aussi pour pouvoir extraire les 2 mots de commande de cette ligne, on n'utilise plus les méthodes de la bibliothèque Scanner mais celles de StringTokenizer. (Avec un changement de la bibliothèque importée au début de la classe).

d) Dans **Game** : Après avoir fini de transvaser son contenu la classe GameEngine, j'ai copié celle de zuul-with-images : Création du moteur et de l'interface puis liaison des 2.
Aussi, j'ai ajouté la spécification du dossier contenant les images.
(C'est plus propre dans un dossier)

e) Dans **GameEngine** :
- J'ai transformé tous les « System.out » par « this.aGui » après avoir ajouté le nouvel attribut.
- J'ai ajouté le chemin relatif de l'image pour chaque room lors de sa création
- j'ai ajouté l'affichage d'image dans différentes fonctions (goRoom, printWelcome, ...)
- j'ai incorporé le changement de processCommand pour lire la String de la ligne et non plus une commande directement. Notamment en supprimant le retour d'un boolen à cette fonction interpretCommand() malgré la contradiction des commentaires présents dans zuul-with-images.
- j'ai édité ma fonction « quit » pour qu'elle utilise la méthode enable() de la GUI.
- j'ai supprimé la fonction play()

f) Comme la fonction printHelp() et look() utilisent encore la fonction printLocationInfo() sans avoir à afficher une nouvelle image, j'ai préféré ajouter la fonction displayLocationImage() qui affiche l'image à condition qu'elle existe. Je l'appelle juste après printLocationInfo() quand nécessaire.

g) Dans **UserInterface** :
- J'ai ajouté l'attribut « aImageFolder » et son assignation dans le constructeur, ainsi qu'une méthode pour changer sa valeur.
Dans la fonction « showImage » j'ai ajouté la prise en compte du dossier pur le chemin des images (le travail avait été commencé par l'auteur, les guillemets étaient simplement vides).
- Dans cette même fonction, j'ai également changé l'échelle des images avec la classe « Image » de java.awt pour qu'elles soient moins grandes, et comme ça j'ai pu aussi agrandir la hauteur de la zone de texte ailleurs dans la classe UserInterface.
- J'ai remplacé les importations de listes de classes par chaque classe utile unique.

Exercice 7.18.8

- Dans UserInterface : j'ai ajouté l'importation de la classe javax.swing.JButton
- Dans le constructeur j'ai ajouté le bouton comme attribut

- Dans createGui : j'ai ajouté la création du bouton qui servira donc à respirer, et je lui ai ajouté un Action Listener.
- J'ai ajouté ce bouton dans la zone Est du panel.
- Dans actionPerformed : je vérifie d'abord si l'objet qui a déclenché l'action est le bouton, auquel cas on fait comme si le programme avait reçu le texte « respirer » pour déclencher la commande associée.
  Autrement, on appelle processCommand() qui lira l'entrée du terminal.
- J'ai fait la même chose pour le bouton « regarder » dans la zone ouest.

Exercice 7.19.2

Déjà réalisé. J'ai simplement renommé le dossier avec une majuscule.
J'ai également déplacé la définition du dossier contenant les images en tant qu'attribut de GameEngine, et assigné dans setGUI().

Exercice 7.20

- J'ai créé une nouvelle classe **Item** avec
  - 2 attributs : le poids et la description
  - leur constructeur et accesseurs
  - une fonction getLongDescription qui renvoie la description complète de l'item comme pour les rooms.
- Dans la classe **Room** j'ai ajouté :
  - un nouvel attribut aItem
  - une fonction pour le définir si voulu.
  - une fonction qui renvoie la string « Auncun objet » ou la decription longue de l'objet s'il existe.
  - dans getLongDescription(), j'ai ajouté la string renvoyée par la fonction précédente.

Exercice 7.21

Déjà réalisé dans l'exercice précédent.
La description de l'item doit être produite par l'item qui va lire ses attributs privés pour créer la String.
La description de la Room est générée dans la classe Room qui lit ses attributs privés (dont l'item)
L'affichage de toutes ces descriptions se fait dans la classe GameEngine qui n'a pas besoin d'accéder aux attributs de l'item ou de la room. On veut découpler les classes au maximum.

Exercice 7.21.1

Dans item, j'ai remplacé la description seule par un nom et une description.
J'ai ajouté 3 objets dans 3 salles.
Je n'ai pas modifié la commande look.

Exercice 7.22

Dans **Room** :
- J'ai remplacé l'attribut aItem par une hashmap d'items ou le nom de l'objet est la clé.
- J'ai changé setItem() par addItem() avec .put
- J'ai changé la fonction getItemString() pour qu'elle liste tous les objets de la hashmap s'il y en a, en

utilisant un Stringbuilder.
Dans **GameEngine**, j'ai remplacé le nom de la fonction pour ajouter un item, et j'ai ajouté une branche en plus de la clé dans l'arbre pour avoir plusieurs items dans une room.

Exercice 7.22.1

J'ai préféré une hashmap pour pouvoir récupérer un item par son nom par exemple si l'utilisateur le désigne par son nom dans l'invite de commande. Sinon il faudrait itérer chaque élément de la collection jusqu'à tomber sur l'objet dont le nom correspond (après y avoir accédé), ce qui serait beaucoup moins optimisé, et ajouterai des lignes de codes.

Exercice 7.22.2

J'avais déjà voulu tester en ajoutant des objets, mais je vais ajouter un objet dans la salle initiale aussi. Disons une buche.

Exercice 7.23

J'ai ajouté un attribut aPreviousRoom au GameEngine, initialisé à null dans createRooms().
Dans goRoom(), avant de changer la valeur de aCurrentRoom, je stocke sa valeur dans aPreviousRoom.

J'ai ajouté le traitement de la commande « retour » au switch case de interpretCommand().
j'ai créé la fonction goBack() qui prend le commande en paramètre. Comme pour quit(), elle vérifie qu'il n'y ait pas de second mot (auquel cas elle prévient le joueur).
Si aPreviousRoom est null, le joeur est informé qu'il ne peut pas revenir en arrière. Autrement, elle remet la salle précédente en tant que courante, avant d'appeler printLocationinfo() et displayLocationImage().
J'ai testé tous les cas possibles de retour et tout fonctionne.

Exercice 7.26

J'ai changé comme indiqué l'attribut précédent par une Stack<Room> déclarée en début de classe puis créée dans le constructeur. J'ai remplacé aux 3 endroits correspondants :
.push() dans goRoom, .empty() pour vérifier si il y a un historique et .pop() pour revenir en arrière.
Tout est parfaitement fonctionnel.

Exercice 7.26.1

J'ai ajouté au PATH le dossier bin de BlueJ, j'ai généré les 2 javadoc (prog et user).
J'ai vu qu'ils attendaient des commentaires pour les attributs alors je les ai ajoutés.
Maintenant il n'y a plus de warnings.

Exercice 7.28.1

Pour implémenter cette nouvelle commande de test, j'ai dû changer plusieurs fonctions au niveau de la lecture des commandes :
Jusqu'à maintenant, dans la classe Parser, la fonction getCommand() en charge d'extraire les 2 mots de la commande, vérifiait si le premier mot appartenait à la liste des commandes valides et si le second mot appartenait à la liste des direction valides (en utilisant les méthodes de la classe CommandWords).

Cela fonctionnait car la seule commande à accepter un second mot était « aller », et mon implémentation (proposée dans le forum sur l'exercice qui traitait du sujet) permettait de renvoyer au joueur 1 des 2 messages suivants en fonction de la situation :
- "Cette direction n'existe pas."
- "Vous ne pouvez pas aller dans cette direction !"

Le problème c'est que maintenant que « test » accepte aussi un second mot je ne pouvais plus me contenter de vérifier si le second mot est valide dans la fonction de Parser qui transforme l'entrée utilisateur en objet commande.
J'ai pallié le problème tout en gardant 2 messages d'erreur différents de la sorte :

- Un peu comme pour la fonction de Parser getCommandsList() qui renvoie la liste des commandes disponibles en appelant une fonction de CommandWords qui crée la String originale : j'ai créé une fonction isDirection() dans Parser qui se contente de renvoyer le résultat de la fonction isDirection() appelée cette fois sur un objet CommandWords créé pour l'occasion.
- Dans ComandWords, j'ai bien-sûr ajouté le mot « test » à la liste des commandes valides
- Dans la classe GameEngine j'ai modifié la fonction goRoom() :
  Anciennement, la fonction vérifiait si le second mot était « invalid » (valeur anciennement assignée au second mot, directement par getCommand() si la direction était invalide).
  Maintenant la fonction vérifie par elle-même si la direction est invalide grâce au booléen pCommand.isDirection(vDirection).

Pour ce qui est du test en lui-même :

Dans la classe GameEngine, j'ai presque recopié l'exemple d'utilisation fourni par la section « Lecture simple de fichiers de texte » de la page « Plus de Technique » dans la liste officielle des exercices :

- J'ai ajouté l'import des classes Scanner, File et FileNotFoundException
- J'ai ajouté le mot clé « test » au switch case de interpretCommand () qui execute alors la fonction « executeTest(vCommand) » décrite ci-dessous :

J'ai donc ajouté la fonction executeTest() qui reprend donc l'exemple de lecture simple de fichier.
J'ai concaténé le second mot de la commande avec « .txt » puis remplacé « **/// traitement de la ligne lue »** l'appel de interpretCommand(vLigne) où vLigne correspond à une nouvelle ligne du fichier .txt.
J'ai aussi remplacé « **/// traitement en cas d'exception »** par un message dans le terminale qui indique que le fichier est introuvable en affichant le nom complet du fichier auquel on a essayé d'accéder.

J'ai aussi créé le fichier essai.txt à la racine avec quelques commandes à tester.
Tout fonctionne correctement lors de son exécution et les erreurs liés à la commande (ex : fichier inexistant) sont bien prises en charges.

Exercice 7.28.2

J'ai créé 3 fichiers :
- court.txt
- optimal.txt : on va chercher la clé dans l'arbre et on va à la porte
- complet.txt : tout est exploré, et les cas d'erreurs sont aussi testés

Exercice 7.29

Dans la classe **Player** :

- J'ai ajouté l'import de la classe Stack.
- J'ai créé l'attribut aName pour stocker le nom du joueur avec son accesseur
- Anciennement dans GameEngine j'y ai déplacé les attributs aCurrentRoom (avec son getter et son setter) et aPreviousRooms .
- J'ai créé un constructeur sans paramètres qui initialise aPreviousRooms ainsi que le nom du joueur en demandant dans un popup le nom au joueur, et si la String est vide ou ne contient que des espaces, on assigne le nom « Link » par défaut.
- J'ai aussi déplacé les assignations de currentRoom de GameEngine dans les 2 fonctions goRoom() (qui prend en paramètre la Room de destination) et goBack() (qui renvoie un booléen pour indiquer si on pouvait effectivement retourner en arrière).

Dans la classe **GameEngine** :

- J'ai supprimé les attributs et l'import déplacés dans Player
- J'ai ajouté un attribut aPlayer et sa fonction setPlayer() (car l'objet Player est créé dans la classe Game). Cette fonction appelle aussi this.aPlayer.setCurrentRoom(this.aStartRoom) car la première pièce est définie dans createRoom() qui est appelée par le constructeur de GameEngine avant qu'on lui donne son joueur. Donc j'ai créé un attribut aStartRoom qui est assigné dans createRooms() et lu dans setPlayer().
- Maintenant, à la fin de createRooms(), on se contente d'assigner la première Room dans l'attribut créé à cet effet.
- Dans goRoom() on appelle simplement la méthode du même nom sur l'attribut aPlayer si le déplacement est valide.
- Dans goBack(), on appelle simplement la fonction du même nom sur l'attribut aPlayer. Cette fonction renvoie « true » ou « false » si l'historique est vide, permettant ainsi à GameEngine d'indiquer si le retour est possible dans la GUI.
- J'ai personnalisé un peu le message de bienvenue et d'au revoir avec le nom du joueur

Dans la classe **Game** :

- Je déclare un nouvel attribut aPlayer et le créée dans le constructeur.
- Comme pour la GUI, je le connecte au GameEngine avec this.aEngine.setPlayer(this.aPlayer)

Exercice 7.30

Dans la classe **Player** j'ai ajouté :

- un attribut aCurrentItem
- addItem() qui assigne l'objet Item passé en paramètre
- removeItem() qui lui donne null comme valeur
- getItem() son accesseur
- hasItem() qui renvoie true ou false pour savoir si le joueur a déjà un objet.

Dans la classe **Room** :

- Une nouvelle fonction removeItem() permet de retirer l'objet de la hashmap via son nom passé en paramètre.
- Une nouvelle fonction getItem() permet de retourner l'objet Item de la pièce, qui n'avait pas besoin d'accesseur auparavant (seulement de renvoyer une String pour l'affichage).

Dans la classe **GameEngine** :

- J'ai ajouté la prise en comptes des 2 nouvelle commandes « prendre » et « poser » dans interpretCommand()
- J'ai donc ajouté la fonction take() qui vérifie d'abord s'il y a bien un 2nd mot à la commande et si le joueur a déjà un Objet.
  Puis qui récupère créé un objet de type Item qui stocke l'objet dans la currentRoom qui porte ce nom.
  Si l'objet se trouve être null, c'est qu'il n'y a pas de tel objet dans la currentRoom, autrement, l'Item est ajouté au Player et retiré de la pièce courante.
  Un message est renvoyé au joueur en fonction de chaque situation.
- Ainsi que la fonction drop() qui ne prend pas de paramètre car il n'y a qu'un objet potentiel à déposer dans la pièce courante.
  Après avoir vérifié l'absence de second mot et le port d'un objet par le Player, drop() ajoute à la pièce courante l'item du joueur et lui retire donc ce dernier.
  Un message est renvoyé au joueur en fonction de chaque situation.

Bien-sûr, j'ai aussi ajouté à CommandWords les 2 mots clé « prendre » et « poser ».

Exercice 7.31

Dans la classe **Player** :

- J'ai ajouté **l'import** de la classe HashMap.
- J'ai remplacé **l'attribut** aCurrentItem (qui ne stockait qu'un seul Item) par une HashMap<String, Item> « aItems », également intitialisée à la fin du **constructeur**.
- J'ai modifié **addItem()** pour qu'elle ajoute l'Item à la HashMap en utilisant le nom de l'Item comme clé avec this.aItems.put().
- J'ai modifié **removeItem()** pour qu'elle prenne maintenant le nom de l'Item en paramètre et utilise .remove() sur la HashMap.
- J'ai modifié **getItem()** pour qu'elle prenne le nom de l'Item en paramètre et renvoie le résultat de .get() sur la HashMap.

- J'ai modifié **hasItem()** pour qu'elle prenne le nom de l'Item en paramètre et utilise .containsKey() pour vérifier si cet Item spécifique est dans l'inventaire.

Dans la classe **GameEngine** :

- J'ai modifié **take()** pour retirer la vérification qui empêchait de prendre un objet si on en portait déjà un. Maintenant le joueur peut prendre autant d'objets différents qu'il le souhaite.
- J'ai modifié **drop()** pour qu'elle demande maintenant quel objet poser via le second mot de la commande (au lieu de simplement déposer l'unique objet porté). Elle vérifie donc qu'un second mot est présent, puis vérifie avec hasItem(vItemName) que le joueur porte bien cet objet spécifique avant de le déposer.

Tout fonctionne correctement : je peux maintenant prendre plusieurs objets différents (bûche, carte, épée, etc.), les déposer individuellement avec "poser nomObjet. La HashMap permet un accès direct par nom d'objet, ce qui est plus efficace qu'une ArrayList qui nécessiterait de parcourir toute la liste.

Exercice 7.31.1

Dans la classe **ItemList** que j'ai créée, j'ai ajouté :

- **L'import** de la classe HashMap
- L'attribut **aItems**, qui est donc une HashMap<String, Item>
- Le **constructeur** pour l'initialiser
- La méthode **add**() qui ajoute à la hashmap l'Item passé en paramètre.
- La méthode **remove**() qui retire de la hashmap l'Item dont le nom est passé en paramètre.
- La méthode **get**() qui retourne l'Item dont le nom est passé en paramètre.
- La méthode booléenne **has**() qui renvoie 1 si l'Item dont le nom est passé en paramètre existe.
- La méthode **isEmpty**() qui vérifie si la liste d'Items est vide.
- La méthode **getItemString**() qui renvoie une String qui liste tous les objets contenus avec leur description, ou « Aucun objet » si elle est vide. J'y ai utilisé StringBuilder.

Dans la classe **Player** :

- J'ai ajouté un attribut « **aInventaire** » de type ItemList, l'ai initialisé dans le **constructeur**.
- Ensuite j'ai remplacé le contenu de toutes les méthodes liées à l'inventaire par une redirection du paramètre vers les méthodes d'ItemList appelées sur l'inventaire.

Dans la classe **Room** :

- J'ai effectué les mêmes changements en ajoutant l'attribut de type ItemList et en modifiant l'implémentations des méthodes associées à cette liste.

Dans la classe **GameEngine**, je n'ai **pas eu besoin de modifier** l'appel des fonctions utilisés depuis Player ou Room car seul leur implémentation a changé en appelant les méthodes de ItemList au lieu de recoder l'opération dans chacune de ces 2 classes.

Exercice 7.32

Dans la classe **Player** j'ai ajouté :

- L'attribut « **aInventoryCapacity** » initialisé à 10 (Kg) dans le constructeur.
- L'attribut « **aInventoryWeigtht** » initialisé à 0 (Kg) dans le constructeur.

- L'incrémentation/décrémentation du poids de l'inventaire dans les méthodes **addItem**() et **removeItem**().
- **L'accesseur** de chacun de ces 2 attributs privés, afin de vérifier qu'on puisse ajouter un item.

Dans la classe **GameEngine** :

- J'ai modifié la méthode **additem**() pour qu'elle vérifie d'abord si l'objet que le joueur veut prendre dépasse la capacité de l'inventaire, et le cas échéant, renvoie un message comme celui-ci :
  « Vous ne pouvez porter que 10.0 kg au maximum.
  Et vous portez déjà 5.0 kg.
  Or cet objet pèse 12.0 kg. »

Exercice 7.33

Dans la classe **CommanWords** :

- J'ai ajouté le mot-clé « inventaire »

Dans la classe **GameEngine** :

- J'ai ajouté la méthode **showInventory**() qui à la fin, appelle this.aPlayer.getInventoryContents() qui elle-même se contente d'appeler getItemsString().
- J'ai ajouté l'association au switch case qui assigne à chaque mot-clé la fonction à exécuter.

Dans la classe **ItemList** :

- J'ai déjà implémenté la méthode **getItemsString()**, utilsée par la méthode getLongDescription de la classe Room.
  Elle utilise un foreach comme suit : for (Item vItem : this.aItems.values()) et un StringBuilder, pour afficher soit « auncun objet » soit « \n – objet1… \n – Objet2… ».

Exercice 7.34

Dans la classe **CommanWords** :

- J'ai ajouté le mot-clé « ingérer »

Dans la classe **GameEngine** :

- J'ai ajouté une fiole d'eau oxygénée comme objet dans la zone de départ.
- J'ai ajouté le traitement de la commande « ingérer » qui renvoie vers la fonction ingest avec la commande en paramètre pour savoir quoi ingérer.
- J'ai ajouté la fonction ingest() qui va d'abord vérifier si un objet est spécifié, puis s'il est dans l'inventaire. Ensuite, un switch case permet d'associer à chaque nom d'objet spécifié, une fonction à exécuter. Des messages sont affichés à chaque fois que l'action n'est pas possible et pourquoi.
- J'ai créé la fonctino drinkH202() qui va doubler la capacité en poids de l'inventaire du joueur en appelant this.aPlayer.doubleInventoryCapacity(), puis expliquer la nouvelle capacité au joueur, puis retirer l'objet de son inventaire.

Dans la classe **Player** :

- J'ai donc ajouté une fonction très simple qui double la valeur de l'attribut aInventoryCapacity.

Exercice 7.34.1

J'ai mis à jour les 3 fichiers .txt en incluant les nouveaux objets et les nouvelles commandes.

Exercice 7.34.2

J'ai regénéré les 2 javadocs.

Exercice 7.42

J'ai décidé de compter uniquement les déplacements.

Dans la classe **GameEngine** :

- J'ai ajouté l'attribut **aMovesCount** initialisé à 0 dans le constructeur.
- J'ai ajouté l'attribut constant **aMaxMoves** fixé à 100 pour pouvoir exécuter le test complet.
- J'ai créé la fonction **countMoves**() pour compter un déplacement de plus puis vérifier si le maximum a été atteint, et, le cas échéant, afficher le Game Over puis quitter le jeu avec this.aGui.enable(false);
- J'ai ajouté l'appel de countMoves() à la fin des fonctions **goRoom**() et **goBack**()

Exercice 7.42.2

Je préfère garder l'IHM actuelle car je suis en retard.

Exercice 7.43

Dans la classe **Player** :

- J'ai ajouté la méthode publique **clearHistory**() qui appelle la méthode .clear() sur la Stack de l'historique des pièces du joueur.

Dans la classe **Room** :

- J'ai créé la méthode booléenne hasExitTo() qui renvoie true si la Room sur laquelle elle est appelée possède une sortie vers la Room passée en paramètre. Son fonctionnement est tout simple, il se base sur la méthode .containsValue() de la classe HashMap :
  `return this.exits.containsValue( pRoom )`

Dans la classe **GameEngine** :

- J'ai **retiré la sortie** Est de la zone Nord.
  La trap door est le passage de la zone Est vers la zone Nord.
- Dans la méthode **goRoom**(), en plus de créer la variable vNextRoom, je crée aussi la variable vCurrentRoom pour garder en mémoire les 2 pièces concernées, puis juste après avoir appelé this.aPlayer.goRoom(vNextRoom), je vérifie si un retour en arrière est possible avec cette condition :
  `if ( ! vNextRoom.hasExitTo( vCurrentRoom )) {}`
  Si elle est vraie, j'appelle clearHistory() sur le joueur.

Exercice 7.44

J'ai ajouté la classe **Beamer** qui hérite Item et qui contient :

- Un attribut **aChargedRoom** de type Room qui contient la salle chargée dans le téléporteur.
- Un **constructeur par** défaut qui utilise d'abord le mot-clé super() pour définir le nom, la description et le poids d'un téléporteur, puis initialise l'attribut aChargedRoom à NULL.
- La méthode **charge**() qui prend en paramètre la Room à charger dans le téléporteur et donne sa valeur à l'attribut aChargedRoom.
- La méthode **trigger**() qui retourne la Room stockée puis la réinitialise à NULL.

Dans la classe **CommandWords** :

- J'ai ajouté les mots-clés « charger » et « déclencher » (à la place de « charge » et « fire »)

Dans la classe **GameEngine** :

- J'ai déclaré un objet de la classe Item « **vTeleporteur** » qui contient un objet de la classe Beamer pour que l'objet puisse être manipulé dans le code comme tous les autres Items (hors actions spécifiques au téléporteur) :
  `Item vBeamer = new Beamer();`
- J'ai ajouté cet objet à la première salle (**zone Sud**) pour faciliter les tests.
- J'ai ajouté la méthode **chargeBeamer**() qui vérifie d'abord si le joueur possède un téléporteur dans son inventaire, puis appelle la méthode charge() sur le téléporteur que porte le joueur, en lui passant en paramètre la salle courante. Pour pouvoir appeler une méthode de Beamer sur un objet déclaré comme Item, j'ai utilisé un cast de type :
  `((Beamer)this.aPlayer.getItem("téléporteur")).charge(this.aPlayer.getCurrentRoom())`
- J'ai ajouté la méthode **triggerBeamer**() qui vérifie d'abord si le joueur possède un téléporteur dans son inventaire, et s'il est bien chargé, puis appelle la méthode trigger() sur le téléporteur que porte le joueur et stocke le résultat de la fonction dans un objet Room utilisé pour se déplacer en exécutant les mêmes commandes que dans goRoom(), mais en forçant la réinitialisation de l'historique pour empêcher d'utiliser la commande « retour » car ce serait une 2e téléportation sans téléporteur.
- J'ai ajouté à la méthode **interpretCommand**() le traitement des 2 commandes « charger » et « déclencher » au switch case qui renvoie vers les 2 fonctions précédentes.

Exercice 7.45

Pour cet exercice, j'ai décidé de redéfinir complètement la manière dont on établit un passage entre 2 Rooms dans la méthode createRooms() de la classe GameEngine.

Voici les 4 types de connexion possible entre 2 Rooms ainsi que la ligne de code que j'ai voulue implémenter pour établir chacun des 4 types de connexion :

- Pour une trap door sans porte :
  `Room.connectRooms(vA, "directionAtoB", vB);`
- Pour un passage bidirectionel entre 2 Rooms sans porte :
  `Room.connectRooms(vA, "directionAtoB", vB, "directionBtoA");`
- Pour une trap door avec porte :
  `Room.connectRooms(vA, "directionAtoB", vB, vCle);`
- Pour un passage bidirectionel entre 2 Rooms avec porte :
  `Room.connectRooms(vA, "directionAtoB", vB, "directionBtoA", vCle);`

Pour cela j'ai utilisé, un peu comme pour les différents constructeurs possibles d'une classe, la redéfinition de la même procédure 4x (Overloading). J'ai préféré garder ces procédures dans Room, et donc les mettre en statique.

Voici donc mes changements nombreux pour cet exercice :

Dans la nouvelle classe **Door** j'ai mis :

- Un attribut « **aKey** » de type Item qui correspond à la clé de cette porte
- Un attribut booléen « **aIsLocked** » qui correspond à l'état de la porte : ouverte ou verrouillée.
- Le constructeur prend nécessairement en paramètre l'Item qui permettra d'initialiser l'attribut correspondant à la clé.
  Aussi j'ai initialisé aIsLocked à true car pourquoi créer une porte ouverte que l'utilisateur fermerait lui-même (sans monstre ni énigme spécialisée) ?
- Une méthode booléenne **isLocked**() qui permet de savoir si la porte est verrouillée.
- Une procédure booléenne **unlock**() qui mets aIsLock à false.
- Une procédure booléenne **lock**() qui mets aIsLock à true.
- Une méthode **getStateDescription**() qui permet d'afficher dans la console l'état de la porte.
- Une méthode **getKey**() qui retourne l'attribut qui est la clé, utilisée pour vérifier si le joueur la possède dans son inventaire.

Dans la classe **Room** pour l'intégration des portes :

- J'ai ajouté un attribut aDoors du type HashMap <String, Door> qui permet d'associer à chaque direction une éventuelle porte.
  (En parallèle de la HashMap aExits qui indique les Rooms associées).
- J'ai initialisé cette HashMap comme aExits dans le constructeur.
- J'ai ajouté la méthode setDoor() qui prend en paramètre une direction (String) et une porte (Door) pour ajouter cette association à la HashMap.
- J'ai ajouté la méthode publique getDoor() qui permet au GameEngine de récupérer la potentielle porte qu'il y a dans une direction donnée (passée en paramètre). Grâce à .get().
- Dans la méthode getExitString() j'ai ajouté "[porte ouverte]" ou "[porte verrouillée]" à la suite de chacune des sorties listées que contient la pièce. Grâce à `vDoor.getStateDescription()`.

Dans la classe **Room** pour l'établissement d'un passage entre 2 Rooms
(Cf. les exemples d'utilisation donnés au début des explications de cet exercice)

- J'ai rendu privée la méthode setExit() car elle n'est plus utilisée que par la classe Room.
- Pour une **trap door sans porte** :
  J'ai ajouté une **1ère fois** la procédure static **connectRooms**() qui prend en paramètres :
  - Room pRoom1
  - String pDirection
  - Room pRoom2
  Ici la méthode se résume à associer pRoom2 en tant que sortie pDirection de la pRoom1
- Pour un **passage bidirectionel** entre 2 Rooms **sans porte** :
  J'ai ajouté une **2ème fois** la procédure static **connectRooms**() qui prend en paramètres :
  - Room pRoom1
  - String pDirection1

- Room pRoom2
- Room pDirection2

Elle répète 2 fois l'opération précédente « .setExit() » : dans un sens puis dans l'autre.

- Pour une **trap door avec porte** :
  J'ai ajouté une **3ème fois** la procédure static **connectRooms**() qui prend en paramètres :
  - Room pRoom1
  - String pDirection
  - Room pRoom2
  - Item pCle

  D'abord elle appelle la fonction du même nom en redirigeant tous les paramètres sauf l'Item.
  Puis créée un objet de type Door en passant le paramètre de type Item correspondant à la clé, en paramètre du constructeur.
  Pour enfin associer cette porte à la direction précisée pour la salle 1.

- Pour un **passage bidirectionnel avec porte** :
  J'ai ajouté une **4ème fois** la procédure static **connectRooms**() qui prend en paramètres :
  - Room pRoom1
  - String pDirection1
  - Room pRoom2
  - String pDirection2
  - Item pCle

  D'abord elle appelle la fonction du même nom en redirigeant tous les paramètres sauf l'Item.
  Puis créée un objet de type Door en passant le paramètre de type Item correspondant à la clé, en paramètre du constructeur.
  Pour enfin associer cette porte, pour chacune des 2 Rooms à sa sortie associée.

Dans la classe **GameEngine**, dans la procédure **createRoom**() :

- J'ai d'abord ajouté une Room vIntérieur qui est la première pièce à l'intérieur des ruines, derrière la porte Sud.
- J'ai converti toutes mes définitions de passages pour utiliser la nouvelle manière de connecter des Rooms (chaque connexion ne fait plus qu'1 ligne au lieu de 2).
- J'ai notamment ajouté une connexion entre la porte de la zone Sud et l'intérieur des ruines, avec la clé en paramètre pour ajouter une porte verrouillée.

Dans la classe **CommandWords** :

- J'ai ajouté les 2 mots-clés « déverrouiller » et « verrouiller ».

Dans la classe **GameEngine** :

- Dans la procédure **interpretCommand**() j'ai ajouté l'association des mot-clés « déverrouiller » et « verrouiller » vers les procédures « unclockDoor » et « lockDoor ».
- Dans la procédure **goRoom**(), j'ai ajouté un test qui vérifie s'il y une porte verrouillée présente dans la direction demandée. Si tel est le cas, on stoppe l'exécution de la procédure en indiquant que la porte est verrouillée.
- J'ai créé une procédure **unlockDoor**() qui prend en paramètre la commande associée.
  Pour déverrouiller une porte, on doit taper « déverrouiller nord » par exemple, si la porte est

dans la direction nord de la pièce courante.
Avant de déverrouiller la porte, 4 tests :
- Est-ce que la commande a un second mot pour préciser la direction ?
- Est-ce qu'il y a une porte dans cette direction ?
- Est-ce que la porte est déjà ouverte ?
- Est-ce que `this.aPlayer.hasItem( vDoor.getKey().getName() )`
Si toutes ces conditions sont validées, on peut donc appeler vDoor.unlock() pour déverrouiller la porte.

- J'ai aussi créé la procédure lockDoor() qui reprend le même fonctionnement pour le verrouillage cette fois.

Exercice 7.45.1

J'ai mis à jour les fichiers test : la victoire pour le moment c'est de passer la porte.
J'ai fait en sorte que si on arrive dans la salle finale, c'est que les tests complet.txt et optimal.txt se sont en principe bien exécutés.

Exercice 7.45.2

J'ai regénéré les javadocs.

Exercice 7.18.5

Pour cet exercice j'ai essayé de discerner s'il valait mieux une HashMap ou une ArrayList, l'un accède aux éléments via une key (comme le nom de la pièce), l'autre accède aux éléments via un index.
En y réfléchissant, la seule situation où une Hashmap serait avantageuse c'est si le joueur doit pouvoir désigner une pièce dans la console. En effet, la recherche par key d'une hamap serait plus efficace qu'une recherche par scan de tous les éléments d'une ArrayList.

Or comme je n'avais pas l'intention de laisser le joueur nommer les pièces dans la console (pour une téléportation par exemple), je suis parti sur une ArrayList, d'autant que choisir une pièce au hasard serait moins complexe pour l'exercice 46, car on manipule déjà des index donc des nombres pour désigner les pièces, et en choisir une à partir d'un nombre aléatoire.
Cependant, en lisant l'énoncé de l'exercicce 7.46.1 et la commande alea, j'ai compris que la désignatin d'une pièce par son nom était imposée pour les tests. J'ai donc changé mon code pour utiliser une HashMap et attribué un nom à chaque pièce.

Voici les changements effectués :

Dans la classe **GameEngine** :

- J'ai **importé** la classe HashMap
- J'ai ajouté l'attribut **aRooms** de type `HashMap<String, Room>`
- J'ai **initialisé** la HashMap dans le constructeur
- J'ai ajouté une procédure **createRoom**() qui à partir de 3 paramètres (nom, description, image), s'occupe de créer un nouvel objet Room en passant en paramètre du constructeur la description et l'image, puis l'ajoute à la HashMap avec pour key son nom.
Enfin, la méthode renvoie la Room ainsi créée pour pouvoir la stocker dans les Rooms dans createRooms().

- Dans **createRooms**(), pour chaque Room, j'ai changé la création d'une nouvelle Room par : `Room vRoomX = this.createRoom("room_x", "dans la room x", "room x.png");`

Exercice 7.46

Pour implémenter la téléportation aléatoire, j'ai créé deux nouvelles classes comme indiqué :

Dans la nouvelle classe **RoomRandomizer**, j'ai mis :

- L'**import** de la classe java.util.Random
- Un attribut **aRooms** de type `HashMap<String, Room>` qui garde la référence vers toutes les pièces du jeu (excepté Transporter)
- Un attribut **aRandom** de type Random pour générer les nombres aléatoires.
- Le **constructeur** qui prend en paramètre la HashMap « aRooms » de GameEngine contenant la liste de toutes les pièces pour initialiser l'attribut aRooms de RoomRandomizer (qui stocke seulement la référence vers la même HashMap sans la recopier bien-sûr).
- La méthode **findRandomRoom**() qui convertit d'abord les valeurs de la HashMap en une ArrayList vRoomsList grâce en donnant en paramètre du constructeur directement aRooms.values() (car on ne peut pas accéder à une HashMap par index numérique), tire un index au hasard via la méthode nextInt() (à laquelle on donne en paramètre vRoomsList.size()), et retourne la pièce correspondante.

Choix d'implémentation :

- J'ai préféré **convertir** la HashMap en ArrayList **à chaque appel** de findRandomRoom() au lieu d'en faire un attribut de la classe, car si jamais GameEngine ajoute une Room à sa HashMap, la liste de RoomRandomizer ne sera plus à jour, même si c'est moins optimisé.
- J'ai préféré garder findRandomRoom() **non-statique** car je veux que TransporterRoom crée un unique objet RoomRandomizer qui lui-même contient la référence vers la HashMpa crée dans GameEngine, au lieu de devoir passer la HashMap en paramètre de la méthode qui serait alors utilisée par TransporterRoom. Je trouve ça plus cohérent en termes de distinction des rôles des classes, et cela permet aussi de créer différents Randomizer comme M.Bureau le mentionnait.

Dans la nouvelle classe **TransporterRoom** qui **hérite** de Room, j'ai mis :

- L'attribut **aRandomizer** de type RoomRandomizer qui sert de générateur de pièce aléatoire autonome.
- Le **constructeur** qui appelle d'abord le constructeur parent via super() (description et image) puis initialise le randomizer avec celui passé en paramètre.
- La méthode redéfinie **getExit**() (Override) qui prend toujours une direction en paramètre pour respecter la signature de la classe mère, mais qui l'ignore totalement pour renvoyer le résultat de `this.aRandomizer.findRandomRoom()`.

Dans la classe **GameEngine** :

- Dans **createRooms**(), j'ai instancié un objet vRandomizer en lui passant en paramètre l'attribut this.aRooms contenant toutes les pièces créées avec createRoom() (Cf. 7.18.5).
- J'ai instancié la TransporterRoom « **vTransporter** » manuellement avec new TransporterRoom() en lui passant en paramètre sa description, son image ainsi que le randomizer, sans passer par

la procédure createRoom() car je ne veux ni l'ajouter à la liste des destinations possibles ni l'adapter procédure au paramètre RoomRandomizer.
- J'ai **connecté** cette pièce à l'intérieur des ruines via Room.connectRooms(). Même si une direction spécifique est définie ("sud"), la redéfinition de getExit rendra le déplacement aléatoire, cela sert juste à donner la possibilité de bouger au joueur.

Exercice 7.46.1

Comme on nous demande d'ajouter une commande de debug qui n'est pas utilisable par le joueur, j'ai voulu l'empêcher d'être affichée dans la liste des commandes quand le joueur tape « aide ».
J'ai donc modifié un peu commandWords pour qu'il puisse valider la commande sans l'afficher :

Dans la classe **CommandWords :**

- J'ai ajouté un 2e tableau de chaînes de caractères comme aValidCommands, mais nommé **aDebugCommands** et contenant la seule commande "alea", pour différencier les commandes de debug des commandes normales.
- J'ai modifié la méthode **isCommand**() en ajoutant un 2e boucle for qui vérifie si la chaîne testée appartient au tableau des commandes valides classiques ET au nouveau tableau des commandes de debug.
- Je n'ai donc pas eu besoin de toucher à la méthode **getValidCommandsString**() car elle lit uniquement le tableau des commandes valides.

Dans la classe **TransporterRoom :**

- J'ai ajouté l'attribut **aForcedRoom** de type Room, initialisé à null dans le constructeur. Il sert à stocker temporairement la destination forcée au lieu d'un choix aléatoire.
- J'ai ajouté la méthode **setForcedRoom**() qui permet de modifier l'attribut précédent en passant en paramètre la Room de destination.
- J'ai modifié la méthode **getExit**() : elle vérifie d'abord si aForcedRoom n'est pas null.
  - Si c'est le cas, elle retourne cette salle et remet immédiatement l'attribut à null pour un usage unique (comme je dois retourner la valeur et la remettre à null en même temps, je suis passé par une variable temporaire).
  - Sinon, elle se comporte comme d'habitude avec le randomizer.

Dans la classe **GameEngine :**

- J'ai ajouté l'attribut booléen **aDebugMode**, initialisé à false dans le constructeur, pour savoir si le jeu est en cours de test automatisé via un fichier ou non.
- Dans la méthode **createRooms**(), j'ai ajouté la TransporterRoom à la HashMap aRooms avec la clé "teleporteur_aleatoire".

Choix d'implémentation : Je voulais mettre la TransporterRoom en attribut de GameEngine pour qu'elle ne fasse pas partie de la hashmap des salles parmi lesquelles RoomRandomizer va piocher aléatoirement (car la Transporter Room pourrait téléporter le joueur dans la même TransporterRoom).
Cependant M.Bureau a insisté sur le fait qu'il faille l'ajouter à la HashMap, alors j'ai suivi le conseil d'un membre du forum qui précisait qu'il avait ajouté une vérification dans le Randomizer pour filtrer le type TransporterRoom pour les exclure les TransporterRoom. J'ai donc dû aussi modifier la classe RoomRandomizer.

- J'ai créé la méthode **bypassRandom**() qui permet de forcer la prochaine Room aléatoire :
    - Elle vérifie d'abord si aDebugMode est vrai.
    - Elle récupère la salle cible demandée (via le second mot de la commande) dans la HashMap aRooms.
    - Elle vérifie si la salle demandée existe.
    - Elle récupère l'objet TransporterRoom dans la HashMap (grâce à sa clé "teleporteur_aleatoire"), effectue un *cast* pour le traiter comme une TransporterRoom, et appelle setForcedRoom() en passant en paramètre la salle cible.
- Dans la méthode **interpretCommand**(), j'ai ajouté le cas "alea" qui redirige vers la nouvelle méthode bypassRandom().
- Dans la méthode **executeTest**(), je passe aDebugMode à true avant la boucle de lecture du fichier, et je le repasse à false une fois le test terminé.

Dans la classe **RoomRandomizer** pour ne pas retomber sur la TransporterRoom en en sortant :

- Dans la méthode findRandomRoom, au lieu de créer une ArrayList qui recopie les valeurs de la HashMap des salles, je crée l'ArrayList vide, puis je la remplis avec un foreach sur aRooms.values(), en ajoutant la Room uniquemen si :
  `! ( vRoom instanceof TransporterRoom )`

Exercice 7.46.3

J'ai fini de tout commenter.

Exercice 7.46.4

J'ai regénéré les javadocs.

Refonte du scénario

Description

Comme je suis en manque de temps (notamment avec les partiels la semaine prochaine), j'ai demandé à M.Bureau si le scénario était important dans le jeu et il m'a dit qu'il fallait au moins 10 pièces dans mon jeu.
J'ai donc simplement gardé mon scénario réduit comme scénario principal après l'avoir un peu étoffé et complété.
J'ai donc modifié le contenu du début de mon rapport pour qu'il décrive le jeu actuel.

Dans mon code

- J'ai modifié le contenu de **createRooms**() pour l'adapter à mon scénario complet.
- J'ai remplacé les 2 **boutons** sur l'interface par « Déclencher » et « Charger » car ils ne demande pas de second mot dans la commande associée.
- J'ai aussi implémenté la condition de **victoire** qui manquait à mon jeu :
  Dans la classe GameEngine, à la fin de la procédure take(), j'ai ajouté une condition qui vérifie si l'objet qui vient d'être ajouté à l'inventaire a pour nom « pyrotaris ». Si tel est le cas, je désactive la console et affiche l'image de victoire ainsi que le message correspondant.
- Dans GameEngine, j'ai ajouté un attribut booléen pour savoir **si le jeu est fini**, et dans interpretCommand(), je bloque toute commande si le jeu est fini, notamment pour ne pas faire changer l'image de victoire ou de défaite via les tests.
- J'ai ajouté les copyrights des 2 images que j'ai prises d'internet et ajouté un écran Game Over.
- J'ai ajouté la possibilité d'ajouter des retours à la lignes et des commentaires (# ..) dans les fichiers tests pour plus de clarté. Donc voivi mes changements dans la boucle while qui regarde chaque ligne dans la fonction executeTest() de GameEngine :
  `if ( Ligne.startsWith( "#" ) || vLigne.trim().isEmpty() ) { continue; }`

Mode d'emploi

Exécuter > `test optimal` pour voir comment gagner en un minimum de temps.

Exécuter > `test complet` pour découvrir tous les scénarios du jeu.

Exécuter > `test gameover` pour voir une situation perdante

Déclaration anti-plagiat

Je déclare n'avoir plagié aucun scénario, ni aucun code déjà existant excepté ceux fournis par les professeurs.