

MAGIC GARDEN

A la quête de l'élixir féerique.



TAKI Yaquine
Groupe 11



Sommaire

I.

I.1) Auteur.....	p.3
I.2) Thème	p.3
I.3) Résumé du scénario	p.3
I.4) Plan	p.4
I.5) Scénario détaillé.....	p.5
I.6) Détail des lieux, items, personnages.....	p.6
I.7) Situations gagnantes et perdantes.....	p.8
I.8) Eventuellement énigmes, mini-jeux, combats, etc.....	p.9
I.9) Commentaires.....	p.10

II. Réponses aux exercices

• Exercice 7.5.....	p.10
• Exercice 7.6.....	p.10
• Exercice 7.7.....	p.11
• Exercice 7.8.....	p.11
• Exercice 7.8.1.....	p.12
• Exercice 7.9.....	p.12
• Exercice 7.10.1.....	p.13
• Exercice 7.10.2.....	p.13
• Exercice 7.11.....	p.13
• Exercice 7.14.....	p.14
• Exercice 7.15.....	p.14
• Exercice 7.16.....	p.14
• Exercice 7.18.....	p.15
• Exercice 7.18.1.....	p.15
• Exercice 7.18.3.....	p.16
• Exercice 7.18.6.....	p.19
• Exercice 7.18.8.....	p.19
• Exercice 7.19.2.....	p.20
• Exercice 7.20.....	p.20



Sommaire

• Exercice 7.21.....	p.21
• Exercice 7.22.....	p.21
• Exercice 7.23.....	p.22
• Exercice 7.26.....	p.23
• Exercice 7.26.1.....	p.24
• Exercice 7.28.1.....	p.24
• Exercice 7.28.2.....	p.25
• Exercice 7.29.....	p.25
• Exercice 7.30/31.....	p.26
• Exercice 7.31.1.....	p.28
• Exercice 7.32.....	p.29
• Exercice 7.33.....	p.30
• Exercice 7.34.....	p.30
• Exercice 7.34.1.....	p.30
• Exercice 7.34.2.....	p.30
• Exercice 7.42.....	p.31
• Exercice 7.42.2.....	p.31
• Exercice 7.43.....	p.31
• Exercice 7.43.1.....	p.32
• Exercice 7.44.....	p.32
III.Déclaration anti-plagiat.....	p.33



RAPPORT PROJET ZUUL

MAGIC GARDEN

I.1 Taki Yaquine

Groupe 11

I.2 Phrase thème :

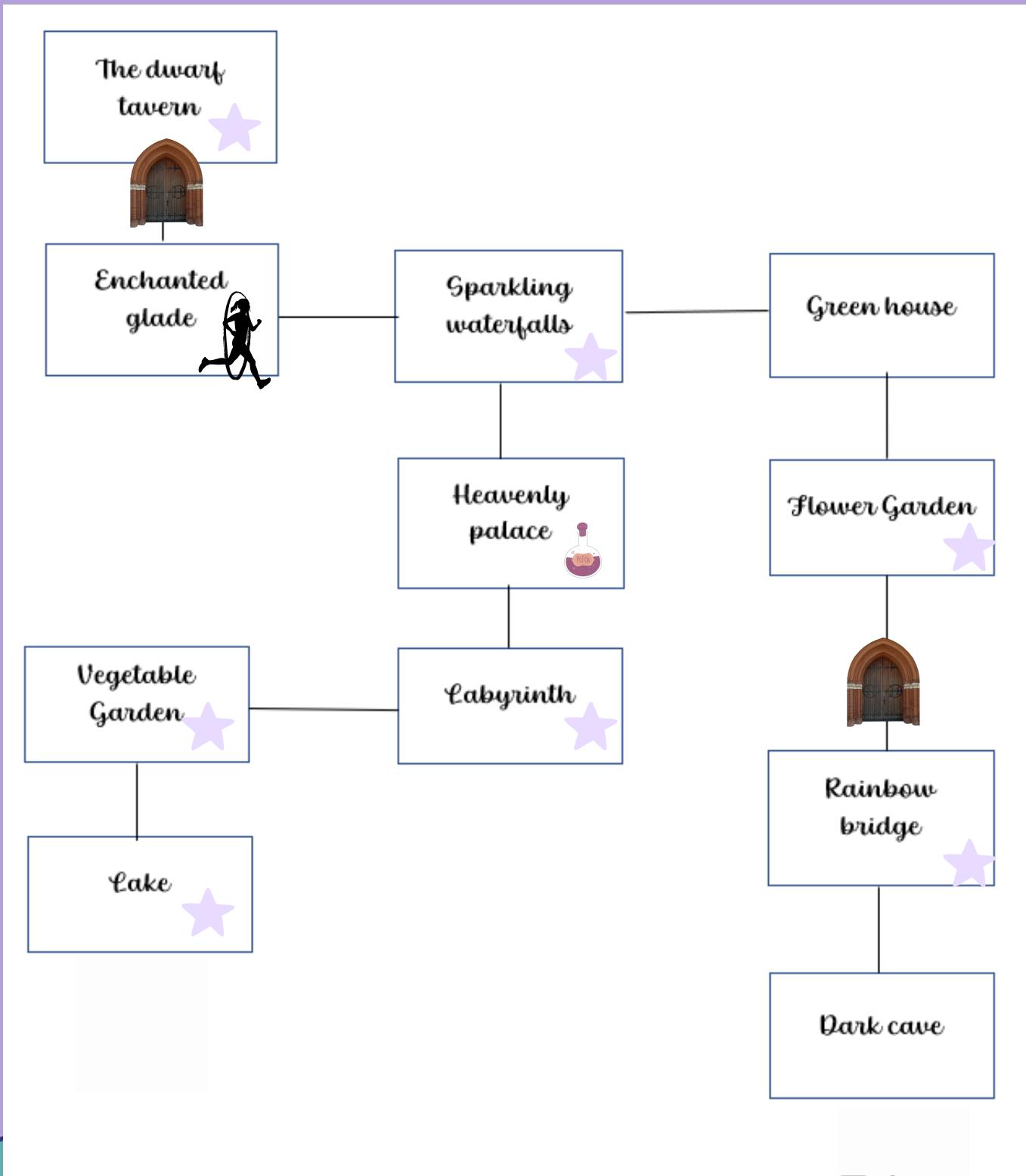
Dans un jardin féerique, une sorcière doit récolter tous les ingrédients nécessaires pour concocter une potion magique qui la métamorphose en fée.

I.3 Résumé du scénario :

La sorcière vieillit et regrette le temps passé. Elle décide donc d'apporter un changement radical en se transformant en fée ! Pour cela, elle devra récolter les ingrédients dispersés dans le jardin féerique pour concocter une potion magique. Mais attention ! La vieille sorcière n'a pas beaucoup de force, elle ne peut donc pas porter les ingrédients pendant un long moment : si la sorcière atteint un certain poids, elle ne pourra plus porter d'ingrédients... et devra donc retourner au palais ! Une fois les ingrédients récoltés, la sorcière devra se diriger vers le palais pour créer la potion et enfin se métamorphoser en fée !



I.4 Plan





RAPPORT PROJET ZUUL

MAGIC GARDEN

I.5 Scénario détaillé

Depuis ses échecs répétés dans des plans machiavéliques visant à retrouver sa prétendue beauté, la sorcière Capucine a été reniée par le Syndicat International des Envoûteurs en raison de son incapacité à ensorceler les autres. Elle a également été accusée de collaborer avec l'ennemi, les fées. Profondément affectée par cette situation, Capucine a sombré dans une grande dépression et a décidé de chercher du réconfort parmi ses amis nains dans leur taverne.

C'est là que son ami Grinchon lui a narré la légende du jardin féerique : « Toute personne dotée de pouvoirs magiques pourrait se transformer en fée en buvant une potion magique ! Mais pour cela, il faut retrouver la liste des ingrédients ! » Intriguée par cette perspective, Capucine a décidé de relever le défi ultime : celui de se transformer en fée à jamais. La liste ainsi que tous les ingrédients sont dissimulés dans l'ensemble du jardin, et elle devra résoudre quelques énigmes pour avancer, car les créatures mystérieuses ne s'aventurent jamais très loin !





RAPPORT PROJET ZUUL

MAGIC GARDEN

I.6 Détails des lieux :

Lieu: **La taverne des nains / Dwarf tavern**

Description : Le joueur débute le jeu à partir de ce lieu, c'est là qu'il devra accomplir sa première mission : retrouver la liste des ingrédients dans le jeu

Personnages : **Grinchon, l'ami nain de Capucine.**

Items : **La liste des ingrédients.**

Lieu: **Clairière enchantée/ enchanted glade**

Description : La clairière enchantée est baignée par la lumière éclatante du matin et encerclée de magnifiques sapin murmurant des indices...

Personnages : **Aucun**

Items : **Cristal (Beamer)**

Lieu: **cascades d'eau scintillante/ sparkling waterfalls**

Description : Une longue cascade d'eau scintillante dévale le flanc de la montagne

Personnages : **Aucun**

Items : **Pierre précieuse de la cascade**

Lieu: **Serre/ Greenhouse**

Description : La serre est rempli de plante exotique

Personnages : **Aucun**

Items : **Aucun**

Lieu: **Jardin des fleurs/ Flower Garden**

Description : Un jardin chatoyant, les fleurs coloré crée une vue magnifique.

Personnages : **Aucun**

Items : **La Rose enchanteresse**





RAPPORT PROJET ZUUL

MAGIC GARDEN

I.6 Détails des lieux :

Lieu: **Palais céleste/ Heavenly palace.**

Description : **Le palais est le lieu finale du jeu, c'est ici que la sorcière devra déposer tous les items/ingrédients retrouvé**

Personnages : **Aucun**

Items : **Aucun**

Lieu: **Labyrinthe/ Labyrinth**

Description : **Le labyrinthe est semé d'embûche.**

Personnages : **Aucun**

Items : **Fougère**

Lieu: **Pont arc-en-ciel/ Rainbow bridge**

Description : **Un gigantesque pont arc-en-ciel**

Personnages : **Aucun**

Items : **morceaux d'arc en ciel**

Lieu: **Potager/ Vegetable garden**

Description : **Un potager rempli de légumes divers et variés**

Personnages : **Aucun**

Items : **Haricot magique**

Lieu: **Lac/ lake**

Description : **Le lac reflète le bleu du ciels étincelant**

Personnages : **Nymphe aquatique**

Items : **bave de crapauds**

Lieu: **Grotte sombre/ Dark cave**

Description : **La grotte est réputé pour les esprits maléfiques qui la hante**

Personnages : **Ogre/ esprits maléfiques**

Items : **Aucun**

I.7 Situations perdues/ Gagnantes

SITUATION PERDANTE

- Lorsque le joueur arrive dans la pièce “Rainbow Bridge”, le joueur devra avoir le beamer chargé dans une pièce avec lui sinon il sera bloquer pour toujours.
- Si le joueur dépasse le temps imparti, le jeu se termine.

SITUATION GAGNANTE

- Réussir à réunir tous les ingrédients dans le palais



II. Réponses aux exercices

EXERCIE 7.5

J'ai mis en place la méthode printLocationInfo() dans la classe Game pour éliminer la duplication de code. En effet, une partie du code servant à fournir des informations sur la position dans le jeu était répétée dans les méthodes goRoom() et printWelcome(). Ainsi, j'ai appelé la méthode printLocationInfo() à la fin de ces deux anciennes méthodes pour connaître les sorties possibles de ma position actuelle.

```
private void printLocationInfo(){
    System.out.println( "You are : " + this.aCurrentRoom.getDescription());
    System.out.print("Exits: ");
    if(this.aCurrentRoom.aNorthExit!=null){
        System.out.print("north ");
    }
    if(this.aCurrentRoom.aEastExit!=null){
        System.out.print("east ");
    }
    if(this.aCurrentRoom.aWestExit!=null){
        System.out.print("west ");
    }
    if(this.aCurrentRoom.aSouthExit!=null){
        System.out.print("south ");
    }
}
```

EXERCIE 7.6

Dans la classe Game et la classe Room, un code est répété, pour éviter cela, on a crée un accesseur getString dans la classe Room, on peut alors supprimer une partie de code dans la procédure goRoom dans la classe Game.

```

public Room getExit(String pDirection){
    if(pDirection.equals("north")){
        return this.aNorthExit;
    }
    if(pDirection.equals("east")){
        return this.aEastExit;
    }
    if(pDirection.equals("south")){
        return this.aSouthExit;
    }
    if(pDirection.equals("west")){
        return this.aWestExit;
    }
    return null;
}

```

EXERCIE 7.7

On veut maintenant que la classe Room produise les informations sur les sorties au lieu de les afficher, et à Game d'afficher ces informations. La fonction `getExitString()` dans la classe room va nous permettre de produire la phrase à afficher dans `printLocationInfo()`, on a donc pu supprimer une grande partie du code.

```

public String getExitString(){
    String vExit="Exits: ";
    if(this.getExit("north")!=null){
        vExit+="north, ";
    }
    if(this.getExit("east")!=null){
        vExit+="east, ";
    }
    if(this.getExit("south")!=null){
        vExit+="south, ";
    }
    if(this.getExit("west")!=null){
        vExit+="west. ";
    }
    return vExit;
}

private void printLocationInfo(){
    System.out.println("You are : " + this.aCurrentRoom.getDescription());
    System.out.println(this.aCurrentRoom.getExitString());
}

```

EXERCIE 7.8

Le but de cet exercice est d'échanger les nombreuses sorties séparé dans chaque méthode par une `HashMap`. Cela nous permet de nous débarrasser des 4 attributs et de simplifier les méthodes dans la classe Room. On a ainsi crée une `HashMap` : "aExits" puis ajouté dans le constructeur naturel Room.

```

*/
import java.util.HashMap;

private String aDescription;
private HashMap <String, Room> aExits;
public Room(final String pDescription){
    this.aDescription=pDescription;
    this.aExits= new HashMap<String, Room>();
}

```

La fonction getExit est maintenant largement plus simple grâce au HashMap qui peut avoir accès au valeurs.

```

public Room getExit(String pDirection){
    return aExits.get(pDirection);
}

```

La procédure setExits n'existe plus, elle est remplacé par la fonction setExit qui ne prends plus que deux paramètres qui permettent de remplir la HashMap.

```

public void setExit( final String pDirection, final Room pSuite){
    aExits.put(pDirection,pSuite);
}

```

Dans la classe Game, on modifie la procédure createRooms avec la nouvelle procédure setExit qui permet de fournir directement les sorties valides pour chaque lieu.

```

vDwarfTavern.setExit("down",vEnchantedGlade);
vEnchantedGlade.setExit("up",vDwarfTavern);
vEnchantedGlade.setExit("east",vSparklingWaterfalls);
vLake.setExit("north",vVegetableGarden);

```

EXERCIE 8.1

Le lieu : La taverne des nains / Dwarf tavern est déjà à une "hauteur" différente des autres lieux (voir le plan page 4).

EXERCIE 7.9

Dans cet exercice, la fonction getExitString est modifiée :
la fonction keySet retourne l'ensemble des clés de la HashMap.
La boucle parcourt toutes les valeurs de la HashMap à travers "keys". On ajoute à la variable vReturnExit la valeur de aExits (les sorties possibles pour chaque lieu) à chaque itération.

```
public String getExitString(){
    String vReturnExit="Exits: ";
    Set<String> keys=aExits.keySet();
    for(String aExits : keys){
        vReturnExit+=" " + aExits;
    }
    return vReturnExit;
}
```

EXERCIE 10.1

La Javadoc est complétée.

EXERCIE 10.2

La Javadoc est générée : <file:///C:/Users/yaqui/OneDrive/IPO/project-zuul/doc/Game.html>

EXERCIE 7.11

Dans la classe Room, on a crée la fonction getLongDescription() afin que le classe produise elle-même la description de la pièce et ainsi réduire le couplage entre la classe Game et Room.

```
public String getLongDescription(){
    return "You are " + aDescription + ".\n" + getExitString();
}
```

On peut alors effectuer les modifications dans la procédure printLocationInfo().

```
private void printLocationInfo(){
    System.out.println(aCurrentRoom.getLongDescription());
}
```

EXERCIE 7.14

Dans la classe CommandWords, on a ajouté la commande “look”. De plus, on déclare le tableau **statique**.

```
private static final String[] aValidCommands={ "quit", "help", "go", "look"};
```

On peut alors effectuer les modifications dans la classe Game. On ajoute la procédure look() et une condition dans la fonction processCommand .

```
private void look(){
    System.out.println(this.aCurrentRoom.getLongDescription());

else if (pCb.getCommandWord().equals("look")){
    if (pCb.hasSecondWord()){
        System.out.println("I don't know how to look at something in particular yet.");
    }
    else{
        this.look();
    }
}
```

EXERCIE 7.15

Dans la classe CommandWords, on a ajouté la commande “eat”.

```
private static final String[] aValidCommands={ "quit", "help", "go", "look", "eat"};
```

On peut alors effectuer les modifications dans la fonction processCommand comme pour la commande look.

```
private void eat(){
    System.out.println("You have eaten now and you are not hungry any more.");
}
```

On ajoute cette condition dans processCommand.

```
else if (pCb.getCommandWord().equals("eat")){
    this.eat();
}
```

EXERCIE 7.16

Dans la classe CommandWords, on a ajouté la procédure showAll qui permet d'afficher toutes les commandes grâce à une boucle qui parcourt le tableau des commandes valides.

```
public void showAll(){
    for(String vCommand: aValidCommands){
        System.out.println(vCommand + " ");
    }
    System.out.println();
}
```

Dans la classe Parser, on ajoute la procédure showCommands.

```
public void showCommands(){
    this.aValidCommands.showAll();
}
```

Puis, on modifie la procédure printHelp dans la classe Game afin de renvoyer les commandes valides .

```
private void printHelp(){
    System.out.println("You are lost. You are alone.");
    System.out.println("You wander in the garden");
    System.out.println("Your command words are:");
    this.aParser.showCommands();
}
```

EXERCIE 7.18

Dans la classe CommandWords, on a remplacé la procédure showAll par la fonction getCommandList qui renvoie toutes les commandes valides.

```
public String getCommandList(){
    String vCommandWord="";
    for(String vCommand: aValidCommands){
        vCommandWord+=vCommand + " ";
    }
    return vCommandWord;
}
```

Dans la classe Parser, on remplace ainsi la procédure showCommands par la fonction getCommands.

```
public String getCommands(){
    return this.aValidCommands.getCommandList();
}
```

Puis, on modifie la procédure printHelp dans la classe Game afin d'afficher les commandes valides.

```
private void printHelp(){
    System.out.println("You are lost. You are alone.");
    System.out.println("You wander in the garden");
    System.out.println("Your command words are:");
    System.out.println(this.aParser.getCommands());
}
```

EXERCIE 7.18.1

La comparaison a été faite.

EXERCIE 7.18.3



La taverne des nains / Dwarf tavern



Potager/ Vegetable garden



Serre/ Greenhouse



Pont arc-en-ciel/ Rainbow bridge



cascades d'eau scintillante/ sparkling waterfalls



Clairière enchantée/ enchanted glade



Grotte sombre/ Dark cave



Palais céleste/ Heavenly palace.



Lac/ lake



Labyrinthe/Labyrinth



Jardin des fleurs/ Flower Garden

EXERCIE 7.18.6

Grâce à la documentation du projet zuul-with-images, on a pu modifier la classe Room, Parser et Game.

Par ailleurs, on a créé deux nouvelles classes : GameEngine et UserInterface.

GameEngine ressemble beaucoup à la classe Game, on a seulement changé certains détails comme deux nouveaux attributs, une procédure setGUI(), on a remplacé processCommand par interpretCommand, et la procédure quit par endGame. De plus, on a légèrement changé la méthode printLocationInfo afin qu'elle puisse afficher aussi l'image de la pièce.

UserInterface est la classe qui permet de créer l'interface du jeu. Pour cela, il faut importer plusieurs classes.

EXERCIE 7.18.8

Dans la classe UserInterface, on a créer un attribut aButton qui correspond à un bouton dans le jeu. Pour le premier bouton j'ai choisis le bouton "eat". Une fois le bouton ajouté, on applique la méthode actionPerformed afin d'exécuter la commande du bouton.

```
private JButton aButton;  
  
//add a button  
this.aButton= new JButton("eat");  
vPanel.add(this.aButton, BorderLayout.WEST);  
  
this.aButton.addActionListener( this );  
  
@Override public void actionPerformed( final ActionEvent pE )  
{  
    // no need to check the type of action at the moment  
    if(pE.getSource()==this.aButton){  
        this.aEngine.interpretCommand("eat");  
    }  
    // because there is only one possible action (text input) :  
    this.processCommand(); // never suppress this line  
} // actionPerformed(.)
```

EXERCIE 7.19.2

Dans cet exercice, on a incorporé les images de chaque pièce dans le jeu en créant un fichier qui contient toutes les images, et en ajoutant leur chemin d'accès grâce notamment au nouveau paramètre dans le constructeur de Room.

```
Room vEnchantedGlade=new Room("in the enchanted glade","image/clairiereenchente.jpg");
Room vLake=new Room("around the lake","image/lac.jpg");
```

EXERCIE 7.20

Pour cet exercice, il faut créer un item. Pour cela il faut d'abord créer une nouvelle classe Item.

```
public class Item
{
    private String aDescription;
    private int aWeight;
    public Item( final String pDescription, final int pWeight){
        this.aDescription= pDescription;
        this.aWeight= pWeight;
    }
    public String getDescription(){
        return this.aDescription;
    }
    public int getWeight(){
        return this.aWeight;
    }
}
```

Puis dans la classe Room, on ajoute un attribut aItem, un accesseur pour les items et un autre pour la description de ces items. Enfin il y a un modificateur pour placer les items dans les différentes pièces.

```
public Item getItem(){
    return this.aItem;
}
public String getItemString(){
    String vD="There is ";
    if(this.getItem()==null){
        return vD + "No item here.";
    }
    else{
        return vD + this.aItem.getDescription();
    }
}
public void setItem(final Item pItem){
    this.aItem= pItem;
}
```

Pour afficher les items présent dans les pièces, il faut ajouter la méthode getItemString() dans la méthode getLongDescription().

```
public String getLongDescription(){
    return "You are " + this.aDescription + ".\n" + this.getExitString() + ".\n" + this.getItemString();
}
```

EXERCICE 7.21

Toutes les informations des items doivent être produite par la classe Item, mais la description de l'item est produite par la classe Room et l'item est affiché par la classe GameEngine.

EXERCICE 7.22

Le but de cet exercice est de pouvoir créer plusieurs items, et pour cela on a besoin de la classe HashMap dans la classe Room; on va créer un HashMap pour produire tous les items :

```
private HashMap <String, Item> aItems;
```

Puis on l'initialise dans le constructeur de Room :

```
this.aItems= new HashMap<String, Item>();
```

On crée aussi un attribut aName dans la classe Item pour simplifier l'appel de l'item dans la classe GameEngine.

```
private String aName;
private String aDescription;
private int aWeight;
public Item(final String pName, final String pDescription, final int pWeight){
    this.aName= pName;
    this.aDescription= pDescription;
    this.aWeight= pWeight;
}
public String getName(){
    return this.aName;
}
```

Pour pouvoir ajouter plusieurs items dans une seule pièce, il va falloir modifier la méthode getItemString() : on a pu garder la première condition if, puis on ajoute la méthode String.join qui est une méthode statique que j'ai trouvé sur internet qui permet de joindre une chaîne de caractère avec un séparateur(ici c'est une virgule). Cela permet de simplifier grandement le code. De plus, pour ajouter les items dans les pièces, il faut créer une procédure addItem qui ajoute un item et son nom dans une pièces.

```

public void addItem(final String pName, final Item pItem){
    this.aItems.put(pName,pItem);
}

public String getItemString(){
    if(this.aItems.isEmpty()){
        return "There are no items here.";
    }
    return "Items in this place: " + String.join(", ",this.aItems.keySet());
}

```

Enfin, on peut créer et afficher les items dans la classe GameEngine.

```

Item vrose= new Item("enchanted rose","You pick the enchanted rose",3);
Item vfougere= new Item("fern","You've finally found the fern leaf ",3);

vLabyrinth.addItem("fern",vfougere);
vDwarfTavern.addItem("list of ingredients",vlistedesingredients);

```

EXERCICE 7.23

Pour créer la commande Back, il faut l'ajouter parmi les commandes valides dans la classe CommandWords.

```
private static final String[] aValidCommands={ "quit", "help", "go", "look", "eat", "back"};
```

Pour créer la procédure back qui permet de revenir dans la précédente pièce, il faut créer un nouveau attribut aPreviousRoom dans la classe GameEngine. Dans la méthode goRoom, on “décale” la position des lieux : la pièce précédente devient la pièce actuelle et la pièce actuelle devient la pièce suivante.

```

if(vNextRoom==null){
    this.aGui.println("There is no door or unknown direction !");
}
else {
    this.aPreviousRoom=aCurrentRoom;
    this.aCurrentRoom = vNextRoom;
    this.printLocationInfo();
}

```

```

private void back(){
    Room vC= this.aCurrentRoom;
    this.aCurrentRoom = aPreviousRoom;
    this.aPreviousRoom=vC;
    this.printLocationInfo();
}

```

```

    else if (vCbS.equals("back")){
        this.back(vCb);
    }
}

```

Et on ajoute la condition ci-dessus dans interpretCommand.

EXERCICE 7.26

On veut cette fois vouloir revenir en arrière sans retomber sur les même pièces à chaque fois, et pour ça, il faut importer la classe Stack et créer un attribut Stack. Dans goRoom, on place dans une pile toutes les pièces traversées par le joueur grâce à push(Room).

```

private Stack<Room> aPreviousRooms;
/**
 * Le constructeur Game appelle la méthode c
 */
public GameEngine(){
    this.createRooms();
    this.aParser= new Parser();
    this.aPreviousRooms= new Stack<Room>();
}

if(vNextRoom==null){
    this.aGui.println("There is no door or unknown direction !");
}
else {
    this.aPreviousRooms.push(this.aCurrentRoom);
    //this.aPreviousRoom=aCurrentRoom;
    this.aCurrentRoom = vNextRoom;
    this.printLocationInfo();
}

```

Puis dans back, on ajoute un paramètre pour analyser si la commande est valide ou non. Si oui, le lieu actuel est supprimé et renvoyé de la liste par pop.

```

private void back(final Command pC){
    Room vC= this.aCurrentRoom;
    if (pC.hasSecondWord()){
        this.aGui.println("The back command doesn't accept a second word.");
    }
    else if(this.aPreviousRooms.empty()){
        this.aGui.println("You can't go back");
    }
    else{
        this.aCurrentRoom=this.aPreviousRooms.pop();
        this.printLocationInfo();
    }
}

```

EXERCICE 7.26.1

La javadoc existe bien : javadoc.exe

Après avoir tapé la commande ci-dessous, la commande javadoc est désormais accessible.

```
SET PATH="C:\Program Files\BlueJ\jdk\bin";%PATH%
```

Pour vérifier que nous avons utilisé le bon chemin d'accès, on tape la commande javadoc --version, et on obtient : javadoc 11.0.14.1

On crée alors un fichier USEJAVA.BAT pour créer un raccourci.

Et enfin, on tape ces deux commandes : javadoc -d userdoc -author -version *.java
javadoc -d progdoc -author -version -private -linksource *.java

EXERCICE 7.28.1

Dans cet exercice, il faut créer la commande test pour tester les différentes commandes et s'assurer du bon fonctionnement du jeu. Pour cela, il faut ajouter la commande "test" dans la classe CommandWords. Puis on crée la méthode test qui lit mon fichier "test.txt" qui contient plusieurs commandes et les teste. La commande tapée doit s'écrire sous la forme : "test test" pour que cela fonctionne.

```
private void test(final Command pCoFichier){  
    if(!pCoFichier.hasSecondWord()){  
        this.aGui.println("What is the file name ?");  
        return;  
    }  
    try{  
        File vFile=new File(pCoFichier.getSecondWord()+".txt");  
        Scanner vTest = new Scanner(vFile);  
        while ( vTest.hasNextLine() ) {  
            String vLine = vTest.nextLine();  
            interpretCommand(vLine);  
        }  
        vTest.close();  
    }  
    catch(final FileNotFoundException pFileNotFoundException){  
        this.aGui.println("An error has occurred");  
    }  
}
```

De plus on importe ces classes très utile pour que la méthode fonctionne.

```
import java.io.File;  
import java.util.Scanner;  
import java.io.FileNotFoundException;
```

EXERCICE 7.28.2

1. Exploration.txt
2. ParcoursIdéal.txt
3. Court.txt

```
go down
look
go east
go east
go south
look
go south
take list_of_ingredients
go down
go east
take stones
go south
drop stones
go south
take fern
go west
take magic_beans
go south
take toad_slime
go north
go east
go north
drop fern
drop magic_beans
drop toad_slime
go north
go east
look
go south
take enchanted_rose
go south
take rainbow_piece
go north
go north
go west
go south
drop enchanted_rose
drop rainbow_piece
quit
help
go down
eat
look
quit
```

EXERCICE 7.29

Dans cet exercice, il faut créer une classe Player et modifier les méthodes qui modifient aCurrentRoom dans la classe GameEngine, c'est-à-dire la méthode goRoom et back. Une partie de leur code va donc être transférée dans la classe Player.

De plus, il faut créer des attributs aNamePlayer, aCurrentRoom et aPreviousRooms et leurs accesseurs.

Enfin, dans GameEngine, il faut remplacer aCurrentRoom par this.aPlayer.getCurrentRoom.

```
public void goRoom(final Command pDir){
    if (!pDir.hasSecondWord()){
        this.aGui.println("Go where ?");
        return;
    }

    if(this.aPlayer.getCurrentRoom().getExit(pDir.getSecondWord())==null){
        this.aGui.println("There is no door!");
        return;
    }

    else {
        this.aPlayer.goRoom(pDir);
        this.printLocationInfo();
    }
}
```

GoRoom et back dans GameEngine

```
private void back(final Command pC){
    if (pC.hasSecondWord()){
        this.aGui.println("The back command doesn't accept a second word.");
    }
    else if(this.aPreviousRooms.empty()){
        this.aGui.println("You can't go back");
    }
    else{
        this.aPlayer.back();
        this.printLocationInfo();
    }
}
```

GoRoom et back dans Player

```
public void goRoom(final Command pDir){  
    Room vNextRoom=this.aCurrentRoom.getExit(pDir.getSecondWord())  
    this.aPreviousRooms.push(this.aCurrentRoom);  
    this.aCurrentRoom = vNextRoom;  
}  
/**  
 * La procédure back permet de revenir dans la pièce précédente.  
 */  
public void back(){  
    this.aCurrentRoom=this.aPreviousRooms.pop();  
}  
}
```

J'ai dû modifier la méthode printLocationInfo car le message "java.lang.NullPointerException" s'affichait, j'ai alors ajouté une condition pour savoir si le joueur est non nul et si la pièce actuelle existe bien.

```
/**  
 * printLocationInfo est une procédure qui affiche les sorties disponibles.  
 */  
private void printLocationInfo(){  
    if (this.aPlayer != null && this.aPlayer.getCurrentRoom() != null) {  
        this.aGui.println(this.aPlayer.getCurrentRoom().getLongDescription());  
        if (this.aPlayer.getCurrentRoom().getImageName() != null) {  
            this.aGui.showImage(this.aPlayer.getCurrentRoom().getImageName());  
        }  
    }  
}
```

EXERCICE 7.30 & 31

Dans cet exercice, il faut créer deux nouvelles commandes : take et drop. Pour cela, on ajoute ces dernières dans la classe CommandWords et on crée pour chacune des commandes, deux méthodes : une dans la classe GameEngine et une autre dans la classe Player.

```

public boolean takeItem(final String pItemName){
    Item vItem= this.aCurrentRoom.getItem(pItemName);

    if(vItem==null){
        return false;
    }
    else{
        this.aCurrentRoom.removeItem(vItem.getName());
        this.aItemsCarried.put(vItem.getName(), vItem);
        return true;
    }
}

```

La méthode take dans GameEngine permet d'afficher les messages en fonction de takeItem.

La méthode takeItem dans Player va retirer l'item pris dans la pièce et va ajouter cet item dans la HashMap : aItemsCarried. Cette HashMap permet de stocker tous les items pris par le joueur

```

public void take(final Command pI){
    if (!pI.hasSecondWord()){
        this.aGui.println("Take what ?");
    }
    String vItemName= pI.getSecondWord();
    Item vItem=this.aPlayer.getCurrentRoom().getItem(vItemName);
    if(this.aPlayer.takeItem(vItemName)){
        this.aGui.println( "You took the " + vItemName );
        this.aGui.println(vItem.getDescription());
    }
    else{
        this.aGui.println("This item isn't here");
    }
}

```

```

public boolean dropItem(final String pItemName){
    Item vItem= this.aItemsCarried.get(pItemName);

    if(vItem==null){
        return false;
    }
    else{
        this.aCurrentRoom.addItem(vItem.getName(), vItem);
        this.aItemsCarried.remove(vItem.getName());
        return true;
    }
}

```

La méthode drop dans GameEngine permet d'afficher les messages en fonction de DropItem.

La méthode dropItem dans Player va ajouter l'item pris dans la pièce et va retirer cet item dans la HashMap : aItemsCarried. Cette HashMap permet de stocker tous les items pris par le joueur .

```

public void drop(final Command pI){
    if (!pI.hasSecondWord()){
        this.aGui.println("Drop what ?");
    }
    String vItemName= pI.getSecondWord();
    if(this.aPlayer.dropItem(vItemName)){
        this.aGui.println( "You drop the " + vItemName );
    }
    else{
        this.aGui.println("You haven't this item");
    }
}

```

La méthode removeItem dans Room permet de supprimer un item d'une HashMap.

```
public void removeItem(final String pName){  
    this.aItems.remove(pName);  
}
```

Enfin, on ajoute ces conditions dans interpretCommand dans GameEngine.

```
else if (vCbS.equals("take")){  
    this.take(vCb);  
}  
else if (vCbS.equals("drop")){  
    this.drop(vCb);  
}
```

EXERCICE 7.3.1

Dans cet exercice, il faut créer la classe ItemList pour éviter la duplication.

```
public class ItemList  
{  
    private HashMap <String,Item> aItemList;  
  
    public ItemList(){  
        this.aItemList= new HashMap<String, Item>();  
    }  
  
    public Item getItem(final String pIn){  
        return this.aItemList.get(pIn);  
    }  
  
    /**  
     * addItem est une procédure qui permet d'ajouter un item dans une pièce.  
     * @param pName est le nom de l'item.  
     * @param pItem est l'objet item.  
     */  
    public void addItem(final String pName, final Item pItem){  
        this.aItemList.put(pName,pItem);  
    }  
  
    public void removeItem(final String pName){  
        this.aItemList.remove(pName);  
    }  
  
    public String getItemString(){  
        if(this.aItemList.isEmpty()){  
            return "There are no items here.";  
        }  
        return "Items in this place: " + String.join(", ",this.aItemList.keySet());  
    }  
}
```

On peut donc retirer les méthodes addItem, removeItem, getItem et getItemString de la classe Room.

Cela implique de créer un attribut itemList dans Room, l'initialiser dans le constructeur et ajouter son accesseur. De plus, la fonction getLongDescription a été légèrement modifiée. Dans Player, on ajoute aussi un attribut itemList aItemCarried pour modifier takeItem et dropItem.

Dans GameEngine, il faut modifier l'ajout des items :

```
vLabyrinth.getItemList().addItem("fern", vFougere);
```

EXERCICE 7.32

Pour pouvoir créer une limite de poids, il faut créer ces deux attributs dans la classe Player :

```
private int aWeightMax=15;  
private int aWeight=0;
```

Puis dans la méthode takeItem et DropItem, on ajoute la condition du poids limite :

```
if(this.aWeight+vItem.getWeight()<=this.aWeightMax){  
    this.aWeight+=vItem.getWeight();  
    this.aCurrentRoom.getItemList().removeItem(vItem.getName());  
    this.aItemCarried.addItem(vItem.getName(), vItem);  
    return true;  
}  
else{  
    return false;  
}
```

TakeItem

```
else{  
    this.aWeight-=vItem.getWeight();
```

DropItem

Dans GamEngine, on peut ajouter cette affichage lorsque le poids maximum est atteint.

```
,  
else{  
    this.aGui.println("This item isn't here or you have reached the maximum weight and cannot carry this item.");  
}
```

EXERCICE 7.33

Dans GamEngine, on crée la méthode Inventory de la commande Inventory, on ajoute la commande dans CommandWords et dans InterpretCommand.

```
public void inventory(final Command pI){  
    this.aGui.println("The items currently carried : "+this.aPlayer.getItemCarried().getItemString()+" . Total Weight: "+ this.aPlayer.getWeight()+"kg");  
}
```

EXERCICE 7.34

Dans Player, on crée la méthode eatBeans qui va multiplier le poids maximum en 2 si le “Magic beans” est mangé.

```
public boolean eatBeans(final String pI){  
    if(pI.equals("magic_beans")){  
        this.aWeight-=this.aItemCarried.getItem(pI).getWeight();  
        this.aItemCarried.removeItem("magic_beans");  
        this.aWeightMax=this.aWeightMax*2;  
        return true;  
    }  
    else{  
        return false;  
    }  
}
```

Dans GameEngine, on modifie la méthode eat afin qu'elle affiche le bon message.

```
private void eat(final Command pM){  
    if(!pM.hasSecondWord()){  
        this.aGui.println("eat what ?");  
        return;  
    }  
    String vItem=pM.getSecondWord();  
    if(!this.aPlayer.eatBeans(vItem)){  
        this.aGui.println("You have eaten now and you are not hungry any more.");  
    }  
    else{  
        this.aGui.println("You ate the magic beans! You became very strong so you can carry more items than before !");  
    }  
}
```

EXERCICE 7.34.1

Les fichiers test sont mis à jour.

EXERCICE 7.34.2

Les javadoc ont été générées.

EXERCICE 7.42

Pour créer une limite de temps, il faut créer un attribut dans GameEngine et l'initialiser dans le constructeur à zéro. Puis dans interpretCommand, le temps est incrémenter à chaque commande entrée. Puis on vérifie que le temps n'est pas écoulé (nombre de commande max =40) sinon, le jeu est terminé et le joueur a perdu.

```
private int aTemps;  
  
this.aTemps+=1;  
if(this.aTemps>=40){  
    this.aGui.println("Game over!");  
    this.endGame();  
    return;  
}
```

EXERCICE 7.42.2

Je garde cette interface graphique pour le moment.

EXERCICE 7.43

Pour avoir une pièce dont une de ses sorties est infranchissable il faut tout d'abord supprimer son accès dans createRooms dans la classe GameEngine. Ensuite, dans la classe Room on crée la méthode isExit grâce à la méthode containsValue qui est utilisée pour vérifier si l'objet passé en paramètre est présent dans la HashMap.

Cette méthode est déjà définie dans la classe HashMap.

```
public boolean isExit(final Room pR){  
    return this.aExits.containsValue(pR);  
}
```

Puis dans la classe player, on modifie la méthode Back qui devient une fonction boolean qui retourne vraie si le joueur peut retourner à la pièce précédente (grâce à la méthode peek de la Stack des pièces précédentes qui permet de voir le dernier élément de la pile, ici c'est la dernière pièce). Si la dernière pièce est une des sorties de la pièce actuelle, celle-ci devient la pièce précédente et son ancienne valeur est supprimé grâce à pop(). effectue la modification habituelle.

Dans cette partie, ma camarade Castillo Thaïs m'a aidé pour la seconde condition de la méthode.

```
public boolean back(){  
    if(this.aPreviousRooms.isEmpty()){  
        return false;  
    }  
    if(this.aCurrentRoom.isExit(this.aPreviousRooms.peek())){  
        this.aCurrentRoom=this.aPreviousRooms.peek();  
        this.aPreviousRooms.pop();  
        return true;  
    }  
    else{  
        return false;  
    }  
}
```

Enfin, on ajoute cette condition pour afficher le message souhaité dans GameEngine.

```
else if(!this.aPlayer.back()){
    this.aGui.println("You can't go back because the climbing plant has blocked the passageway");
}
```

EXERCICE 7.43.1

La javadoc est générée.

EXERCICE 7.44

Pour créer un beamer, qui est une sorte d'item, il faut créer tout d'abord la classe Beamer. On ajoute 2 attributs. Le aRoomCharged est la pièce qui sera mémorisée dans le jeu, et aIsCharged est le booléen qui informe sur l'état d'avancement de la téléportation. Enfin, on ajoute leur accesseur ainsi que deux méthodes essentielles : Charge et Fire.

```
public class Beamer extends Item
{
    private Room aRoomCharged;
    private boolean aIsCharged;
    public Beamer(final String pName, final String pDescription, final int pWeight, final Room pRoom){
        super(pName,pDescription, pWeight);
        this.aRoomCharged= null;
        this.aIsCharged=false;
    }

    public Room getRoomCharged(){
        return this.aRoomCharged;
    }

    public boolean isCharged(){
        return this.aIsCharged;
    }
}
```

Pour la méthode Charge, on stocke simplement la pièce entrée en paramètre dans l'attribut aRoomCharged et on place le boolean à true.

```
public void charge(final Room pRoomCharged){
    this.aRoomCharged= pRoomCharged;
    this.aIsCharged=true;
}

public void fire(){
    if (this.aIsCharged){
        this.aIsCharged=false;
    }
}
```

Pour la méthode fire, on vérifie que le beamer a bien été chargé et on place le boolean à false.

Dans la classe GameEngine, on crée un attribut aBeamer, et on initialise un beamer qui s'appelle "crystal" :

```
this.aBeamer= new Beamer("crystal"," With this crystal, you can teleport anywhere by charging it !",3,vEnchantedGlade);
```

```

private void charge(final Command pC){
    if (!pC.hasSecondWord()){
        this.aGui.println("Which beamer must be charged?");
        return;
    }
    String vBeamerName= pC.getSecondWord();
    Room vCurrentRoom= this.aPlayer.getCurrentRoom();
    Item vItem= this.aPlayer.getItemCarried().getItem(vBeamerName);
    if(vItem==null){
        this.aGui.println( "You didn't take the beamer !");
        return;
    }
    else{
        Beamer vBeamer= (Beamer) vItem;
        vBeamer.charge(this.aPlayer.getCurrentRoom());
        this.aGui.println( "You have charged the beamer !");
        return;
    }
}

```

Dans la classe Games engine la méthode charge est constitué de trois conditions la première vérifie qu'il y a un second mot, la seconde vérifie que l'item a été porté par le joueur, si oui l'item est converti en beamer et nous pouvons ainsi charger le Beamer.

```

public void fire(final Command pC){
    if (!pC.hasSecondWord()){
        this.aGui.println( "Which beamer must be fired?");
        return;
    }
    String vBeamerName= pC.getSecondWord();
    Beamer vBeamer= (Beamer) this.aPlayer.getItemCarried().getItem(vBeamerName);

    if(vBeamer==null){
        this.aGui.println( "You didn't take the beamer and you didn't charge it!");
        return;
    }
    this.aPlayer.setCurrentRoom(vBeamer.getRoomCharged());
    this.aBeamer.fire();
    this.aPlayer.dropItem(vBeamerName);
    this.printLocationInfo();
    this.aGui.println("Well done! you've saved time !");
    return;
}

```

Pour la méthode fire il y a deux conditions la première vérifie encore qu'il y a bien un second mot, la deuxième que l'objet a bien été pris par le joueur, et la troisième condition vérifie que le Beamer a été chargé. S'il a été chargé, modifier la pièce actuelle du joueur, on retire l'item des objets portés par le joueur et on applique la méthode fire de la classe Beamer.

IV. Déclaration anti-plagiat

- Je déclare n'avoir rien copié.
- Toutes les images proviennent de : <https://app.leonardo.ai/>
- Exercice 7.22 source : <https://codegym.cc/fr/groups/posts/fr.864.mthode-java-string-join>
- J'ai été aidé par mes camarades Castillo Thaïs et Oulboub Sarah pour les deux derniers exercices.