

Rapport Hellfire Odyssey

Rapport :

Hellfire Odyssey + lien page web

(A3P(AL) 2022/2023 G6)

I.A) Auteur

Aloïs Fournier

I.B) Thème (phrase-thème validée)

Défier les 9 cercles de l'enfer afin d'accéder au paradis

I.C) Résumé du scénario (complet)

à faire

I.D) Plan (complet)

A incorporer dans le rapport

I.F) Détail des lieux, items, personnages

I.G) Situations gagnantes et perdantes

I.H) Eventuellement énigmes, mini-jeux, combats, etc.

I.I) Commentaire (ce qui manque, reste à faire, ...)

Le Scénario, les situations gagnantes et perdantes et le plan ne sont pas encore incluses.

II. Réponses aux exercices (à partir de l'exercice 7.5 inclus)

7.5) La duplication de code étant l'ennemi numéro 1 en IGI, on a créé une méthode `printLocationInfo()` qui permet de connaître les informations de la pièce courante (description et sorties)

7.6) dans `setExits` on effectue une série de checks pour déterminer si les sorties sont null ou pas, ce qui est redondant et a pu être fixé par l'ajout d'une méthode `getExit()` qui retourne la room correspondant au paramètre `pDirection` qui est égal à l'un des 4 points cardinaux.

7.7)

Afin d'éviter la duplication de code, il est possible d'améliorer l'efficacité de la méthode `"printLocationInfo()"`. Pour ce faire, on crée une nouvelle méthode appelée `"getExitString()"`, qui permet d'obtenir une chaîne de caractères déjà formatée indiquant les sorties de la pièce. Cela permet de donner cette responsabilité à la classe `"Room"`, qui en est directement concernée, et qui est donc plus adaptée, car la classe `"Game"` est le moteur de jeu.

On modifie ensuite `printLocationInfo()` pour inclure les changements.

7.8)

On remplace les 4 attributs par une `HashMap` qui facilitera l'ajout d'autres directions.

Suite à ce changement, il faut adapter le constructeur naturel

ainsi que la procédure `getExit()` car les attributs ont changés

On doit également changer la méthode `setExits()`

On doit donc également réécrire la création de la sortie des Rooms.

Afin de pallier au problème que le programme n'affiche plus que la direction est inconnue mais uniquement qu'il n'y a pas de porte, on crée un nouvel attribut et une nouvelle méthode statique dans la classe `CommandWords`.

On l'intègre maintenant dans `goRoom()` afin d'avoir les bons messages.


7.8.1)

On commence par adapter nos changements sur `CommandWords` en rajoutant « up » and « down ».

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains a single line of Java code.

```
1 private static final String[] aValidDirection = {"north", "south", "east", "west", "up", "down"};
```

On modifie donc l'altitude des salles de fin, l'enfer est en bas et le paradis est en haut :

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains three lines of Java code.

```
1 vLakeCocytus.setExit("up", vParadise);
2 vLakeCocytus.setExit("down", vHell);
3
```

7.9)

On modifie la méthode `getExitString()` afin d'utiliser un `Set` qui est peuplé des keys (identifiants) de la `HashMap` créée dans les exercices précédents

7.10.1)

On annote toutes les méthodes et classes créées pour la javadoc

7.10.2)

On génère donc la javadoc avec BlueJ

7.11)

On se propose de remplacer la duplication de code quand on a besoin d'afficher la description complète (description, sorties...)

On crée donc une méthode `getLongDescription()` qui va se charger de remplacer ces deux instructions.

```
/**
 * Retourne la description complète de la salle
 * @return Description complète de la salle
 */
public String getLongDescription() {
    return "You are " + this.aDescription + ".\n" + this.
getExitString();
}
```

7.14) On crée une nouvelle commande `look`, qui consiste à lister les sorties possibles de la `Room` courante, on intègre cette nouvelle commande aux classes

```
/**
 * Traite la commande
 * @param pCommand La commande
 * @return true si la commande est process, false sinon
 */
private boolean processCommand(final Command pCommand) {
    switch (pCommand.getCommandWord()) {
        case "go":
            this.goRoom(pCommand);
            return false;
        case "help":
            System.out.println("Available commands: go, quit, help");
            return false;
        case "quit":
            return this.quit(pCommand);

        case "look":
            this.look();
            return false;
        default:
            System.out.println("I don\'t know what you mean...");
            return false;
    }
}
```

```

/**
 * Traite la commande
 * @param pCommand La commande
 * @return true si la commande est process, false sinon
 */
private boolean processCommand(final Command pCommand) {
    switch (pCommand.getCommandWord()) {
        case "go":
            this.goRoom(pCommand);
            return false;
        case "help":
            System.out.println("Available commands: go, quit, help");
            return false;
        case "quit":
            return this.quit(pCommand);

        case "look":
            this.look();
            return false;
        default:
            System.out.println("I don\'t know what you mean...");
            return false;
    }
}

```

7.15)

On ajoute une nouvelle commande, j'ai choisi d'ajouter la commande *pray* qui permettra dans le futur du projet de nouvelles fonctionnalités

```

/**
 * Traite la commande
 * @param pCommand La commande
 * @return true si la commande est process, false sinon
 */
private boolean processCommand(final Command pCommand) {
    switch (pCommand.getCommandWord()) {
        case "go":
            this.goRoom(pCommand);
            return false;
        case "help":
            System.out.println("Available commands: go, quit, help");
            return false;
        case "quit":
            return this.quit(pCommand);

        case "look":
            this.look();
            return false;
        default:
            System.out.println("I don't know what you mean...");
            return false;
    }
}

```

```

public CommandWords()
{
    this.aValidCommands = new String[5];
    this.aValidCommands[0] = "go";
    this.aValidCommands[1] = "help";
    this.aValidCommands[2] = "quit";
    this.aValidCommands[3] = "look";
    this.aValidCommands[4] = "pray";
} // CommandWords()

```

7.16)

On ajoute maintenant une méthode permettant de lister toutes les commandes dans la classe CommandWords (showAll) :

```

public void showAll() {
    for (String command : this.aValidCommands) {
        System.out.print(command + " ");
        System.out.println();
    }
}

```

on l'appelle également dans la classe Parser avec la méthode showCommands :

```

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

```

7.18)

On procède donc à la simplification de l'affichage des commandes disponibles suivant le modèle demandé (une classe prépare l'information, la seconde la retourne et la dernière l'affiche) :

On commence par la classe CommandWords, on crée un getter getCommands() qui retourne l'attribut aValidCommands qui contient nos commandes

```

1
2 public String[] getCommands() {
3     return this.aValidCommands;
4 }

```

On crée ensuite dans la classe Parser la méthode getCommandsList() qui va fabriquer le String affiché dans Game :

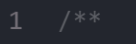


```

1 public String getCommandList() {
2     String pCommandList = "Your commands are: \n ";
3     for ( String vCommand : this.aValidCommands.getCommands()) {
4         pCommandList += vCommand + " ";
5     }
6     return pCommandList;
7 }

```

On appelle finalement cette méthode en utilisant l'attribut aParser dans la classe Game :



```

1 /**
2  * Affiche l'aide du jeu
3  */
4 private void prinHelp() {
5     System.out.println(this.aParser.getCommandList());
6 }

```

7.18.1)

Je n'ai apporté aucun correctif majeur en comparaison avec zuul-better

7.18.6)

Les correctifs suggérés ont été apportés, aucune error/warning ne subsiste.

7.18.8)

On modifie la classe UserInterface en y ajoutant un attribut privé aButton, on l'initialise dans la méthode createGUI() et on le place à l'est du layout. On ajoute également un ActionListener qui ferme le jeu lorsque le bouton est cliqué :



```
1 private void createGUI()
2 {
3     this.aMyFrame = new JFrame( "Hellfire Odyssey" ); // change the title
4     this.aEntryField = new JTextField( 34 );
5     this.aButton = new JButton( "Quit" );
6
7     this.aLog = new JTextArea();
8     this.aLog.setEditable( false );
9     JScrollPane vListScroller = new JScrollPane( this.aLog );
10    vListScroller.setPreferredSize( new Dimension(200, 200) );
11    vListScroller.setMinimumSize( new Dimension(100,100) );
12
13    JPanel vPanel = new JPanel();
14    this.aImage = new JLabel();
15
16    vPanel.setLayout( new BorderLayout() ); // ==> only five places
17    vPanel.add( this.aImage, BorderLayout.NORTH );
18    vPanel.add( vListScroller, BorderLayout.CENTER );
19    vPanel.add( this.aEntryField, BorderLayout.SOUTH );
20    vPanel.add( this.aButton, BorderLayout.EAST );
21
22    this.aMyFrame.getContentPane().add( vPanel, BorderLayout.CENTER );
23
24    // add some event listeners to some components
25    this.aEntryField.addActionListener( this );
26    this.aButton.addActionListener( new ActionListener() {
27        public void actionPerformed((ActionEvent e) { System.exit(0); }
28    } );
29
30    // to end program when window is closed
31    this.aMyFrame.addWindowListener( new WindowAdapter() {
32        public void windowClosing(WindowEvent e) { System.exit(0); }
33    } );
34
35    this.aMyFrame.pack();
36    this.aMyFrame.setVisible( true );
37    this.aEntryField.requestFocus();
38 } // createGUI()
39
```


7.22) Dans cet exercice on créer une classe item tel que demandé dans l'énoncé, on l'incorpore également dans la classe room avec la méthode addItem() car on ne veux pas nécessairement qu'il y ai un item par salle.



```

1  /**
2   * This class represents an item in the game.
3   */
4  public class Item {
5      private String aDescription;
6      private int aWorth;
7
8      /**
9       * Constructor for objects of class Item
10      */
11     public Item(final String pDescription, final int pWorth) {
12         this.aDescription = pDescription;
13         this.aWorth = pWorth;
14     }
15
16     /**
17      * @return the description of the item
18      */
19     public String getDescription() {
20         return this.aDescription;
21     }
22
23     /**
24      * @return the worth of the item
25      */
26     public int getWorth() {
27         return this.aWorth;
28     }
29 }
30

```

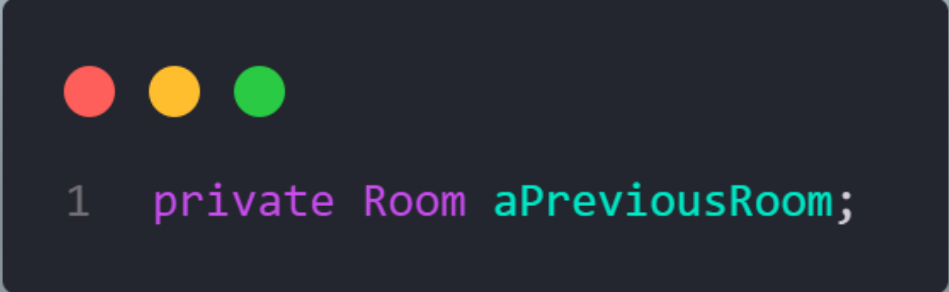


```

1  public void addItem(final String pItemName, final int pWorth) {
2      this.aItem.put(pItemName, new Item(pItemName, pWorth));
3  }

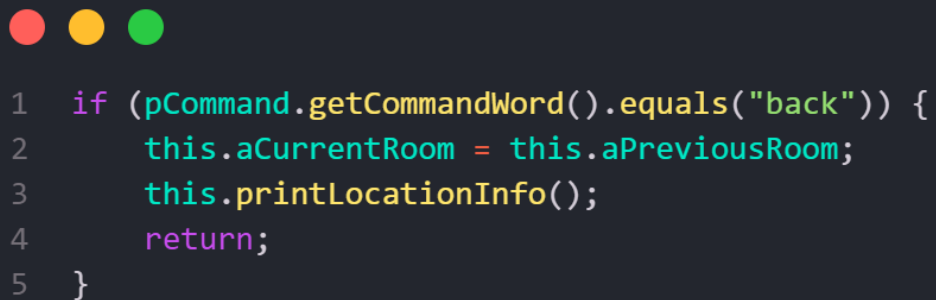
```

On ajoute un attribut `aPreviousRoom` qui contient la room d'avant.




```
1 private Room aPreviousRoom;
```

On implémente dans `goRoom` une fonctionnalité pour revenir a la salle précédente :



```
1 if (pCommand.getCommandWord().equals("back")) {  
2     this.aCurrentRoom = this.aPreviousRoom;  
3     this.printLocationInfo();  
4     return;  
5 }
```

7.26) On modifie l'attribut en lui imputant un type `Stack` en important `java.util.Stack`, On modifie ensuite la méthode `goRoom` pour utiliser la totalité des features proposé par le `Stack` :



```

1  import java.util.Stack;
2
3  /**
4   * Classe Game - le moteur du jeu d'aventure Zuul.
5   *
6   * @author Aloïs Fournier
7   */
8  public class GameEngine
9  {
10     private Room aCurrentRoom;
11     private Stack<Room> aPreviousRooms;
12     private Parser aParser;
13     private UserInterface aGui;

```



```

1
2  if (pCommand.getCommandWord().equals("back")) {
3      this.aCurrentRoom = this.aPreviousRooms.get(this.aPreviousRooms.size() - 1);
4      this.aPreviousRooms.remove(this.aPreviousRooms.size() - 1);
5      this.printLocationInfo();
6      return;
7  }

```

7.26.1) On régénère la JavaDoc en utilisant les deux commandes données. Elles se trouvent donc sous doc/

7.28.1) On ajoute la méthode test.



```

1  private void test(String pFile)
2      throws IOException {
3      if (pFile == null) {
4          throw new IllegalArgumentException("File name cannot be null");
5      } else if (pFile.contains(".txt")) {
6          pFile = pFile.substring(0, pFile.length() - 4);
7      }
8
9      Path path = Paths.get("./" + pFile + ".txt");
10
11     List<String> read = Files.readAllLines(path);
12
13     for (String line : read) {
14         this.interpretCommand(line);
15     }
16 }

```

7.28.2)

Je n'ai ajouté que court.txt et long.txt car mon jeu n'est pas fini et je n'ai donc pas de route qui explore toutes les possibilités.

J'ai cependant ajouté une commande respawn pour pouvoir tester les fichiers plus rapidement sans avoir à relancer le jeu/naviguer jusqu'à la salle de départ.



```

1  public void respawn() {
2      this.aPlayer.setCurrentRoom(this.spawnRoom);
3      this.printLocationInfo();
4      this.aGui.println("You respawned");
5  }
6

```

This.spawnRoom étant égal à vLimbo

7.29)

Avec l'ajout de la classe player, on peut maintenant clean le code de la classe GameEngine qui est surchargée. La majorité des commandes vont donc être réécrites dans la nouvelle classe tout en les conservant dans la classe GameEngine :

```
1  /**
2   * Change de salle
3   * @param pRoom La nouvelle salle
4   */
5  private void goRoom(final Command pCommand) {
6      if (!pCommand.hasSecondWord()) {
7          this.aGui.println("Go Where ?");
8          return;
9      }
10
11     if (!CommandWords.isDirection(pCommand.getSecondWord())) {
12         this.aGui.println("Unknown direction !");
13         return;
14     }
15
16     String vDirection = pCommand.getSecondWord();
17
18     if (this.aPlayer.getCurrentRoom().getExit(vDirection) == null) {
19         this.aGui.println("There is no door !");
20         return;
21     }
22
23     this.aPlayer.goRoom(vDirection);
24     this.printLocationInfo();
25 }
26
```

Exemple de Méthode dans GameEngine ci-dessus, méthode dans Player ci-dessous.



```
1
2 public void goRoom(final String pNextRoom) {
3     this.aPreviousRooms.push(this.aCurrentRoom);
4     Room vNextRoom = this.aCurrentRoom.getExit(pNextRoom);
5     this.aCurrentRoom = vNextRoom;
6 }
```

Suivant le design imposé, un joueur effectue lui-même les actions donc on refactor les commandes que le joueur peut faire (look, go...) dans la classe Player.



```

1  import java.util.Stack;
2
3  public class Player {
4      private String aName;
5      private Room aCurrentRoom;
6      private Stack<Room> aPreviousRooms;
7
8      public Player(final String pName, final Room pCurrentRoom) {
9          this.aCurrentRoom = pCurrentRoom;
10         this.aName = pName;
11         this.aPreviousRooms = new Stack<Room>();
12     }
13
14     // Commands
15
16     public void goRoom(final String pNextRoom) {
17         this.aPreviousRooms.push(this.aCurrentRoom);
18         Room vNextRoom = this.aCurrentRoom.getExit(pNextRoom);
19         this.aCurrentRoom = vNextRoom;
20     }
21
22     public void back() {
23         this.aCurrentRoom = this.aPreviousRooms.pop();
24     }
25
26     // Getters
27     public String getName() {
28         return this.aName;
29     }
30
31     public Room getCurrentRoom() {
32         return this.aCurrentRoom;
33     }
34
35     public Stack<Room> getPreviousRooms() {
36         return this.aPreviousRooms;
37     }
38
39
40
41 }
42

```



On ajoute dans la classe Player et GameEngine une commande take et une commande drop.

Le fonctionnement est simple pour les deux commandes. Pour chacune de celles-ci, GameEngine effectue les checks nécessaires pour ne pas « déranger » Player inutilement puis appelle la méthode correspondante à l'aide de l'attribut aPlayer. Ci-dessous les méthodes take et drop dans les deux classes :

```
1 private void take(final Command pCommand) {
2     if (!pCommand.hasSecondWord()) {
3         this.aGui.println("Take What ?");
4         return;
5     }
6
7     String vItemName = pCommand.getSecondWord();
8
9     if (!this.aPlayer.getCurrentRoom().hasItem(vItemName) && !this.aPlayer.hasItem(vItemName)) {
10        this.aGui.println("There is no " + vItemName + " in this room.");
11        return;
12    }
13
14    this.aPlayer.take(vItemName);
15
16 }
17
```

```
1 public void take(final String pItemName) {
2     this.aInventory = this.aCurrentRoom.getItem(pItemName);
3     this.aCurrentRoom.removeItem(pItemName);
4 }
```

On ajoute également les méthodes removeItem, getItem, hasItem dans la class Room. Le fonctionnement est implicite.



```

1 public void drop(final String pItemName) {
2     this.aCurrentRoom.addItem(this.aInventory.getName(), this.aInventory);;
3     this.aInventory = null;
4 }
5

```




```

1 /**
2  * Drop an item
3  * @param pCommand
4  */
5 private void drop(final Command pCommand) {
6     if (!pCommand.hasSecondWord()) {
7         this.aGui.println("Drop What ?");
8         return;
9     }
10
11     String vItemName = pCommand.getSecondWord();
12
13     if (!this.aPlayer.hasItem(vItemName)) {
14         this.aGui.println("You don't have " + vItemName + " in your inventory.");
15         return;
16     }
17
18     this.aPlayer.drop(vItemName);
19 }
20

```

7.31)

J'ai décidé d'utiliser les HashMaps pour l'inventaire, la flexibilité de la taille et la possibilité de chercher une valeur par clé est très utile :



```

1 public void take(final String pItemName) {
2     this.aInventory.put(pItemName, this.aCurrentRoom.getItem(pItemName));
3     this.aCurrentRoom.removeItem(pItemName);
4 }
5
6 public void drop(final String pItemName) {
7     this.aCurrentRoom.addItem(this.aInventory.get(pItemName).getName(), this.aInventory.get(pItemName));;
8     this.aInventory.remove(pItemName);
9 }
10

```

7.31.1) Afin de mutualiser les listes d'items, on crée une nouvelle classe. J'ai ajouté quelques méthodes utilitaires nécessaires à la manipulation correcte des données de cette classe.

```
1  import java.util.HashMap;
2
3  public class ItemList {
4      private HashMap<String, Item> aItemList;
5
6      public ItemList() {
7          this.aItemList = new HashMap<String, Item>();
8      }
9
10     public void addItem(final String pItemName, final Item pItem) {
11         this.aItemList.put(pItemName, pItem);
12     }
13
14     public Item getItem(final String pItemName) {
15         return this.aItemList.get(pItemName);
16     }
17
18     public void removeItem(final String pItemName) {
19         this.aItemList.remove(pItemName);
20     }
21
22     public boolean hasItem(final String pItemName) {
23         if (this.aItemList.containsKey(pItemName)) {
24             return true;
25         } else {
26             return false;
27         }
28     }
29
30     public String getItemString() {
31         String items = "Items: ";
32         Set<String> keys = this.aItemList.keySet();
33
34         for(String itemKey : keys) {
35             items += itemKey + " ";
36         }
37         return items;
38     }
39 }
```

On incorpore ensuite l'utilisation de cette classe dans Player et Room.

Exemple de changements dans Room :

```
108 public String getItemString() {
109     return this.aItemList.getItemString();
}

Room.java Git local working changes - 8 of 8 changes
107 107
108 108 public String getItemString() {
109     String items = "Items: ";
110     Set<String> keys = this.aItem.keySet();
111
112     for(String itemKey : keys) {
113         items += itemKey + " ";
114     }
115     return items;
109     return this.aItemList.getItemString();
116 110 }
117 111 } // Room
118 112
```

7.32) Dans cet exercice on ajoute deux attributs dans la classe Player, puis on modifie la déclaration de chaque item pour leur attribuer un poids (Cela nécessite également une modification de la classe Item). On ajoute enfin les modifications nécessaires dans take() et drop() pour prendre en compte le changement de poids. J'ai ici choisi d'utiliser un poids maximum aléatoire.

```
1 private int aMaxWeight;
2 private int aCurrentWeight;
3
4 public Player(final String pName, final Room pCurrentRoom) {
5     this.aCurrentRoom = pCurrentRoom;
6     this.aName = pName;
7     this.aPreviousRooms = new Stack<Room>();
8     this.aInventory = new ItemList();
9     this.aMaxWeight = new Random().nextInt(20 + 10) + 10;
10    this.aCurrentWeight = 0;
11 }
```

```
1 private int aWeight;
2
3 /**
4  * Constructor for objects of class Item
5  */
6 public Item(final String pName, final String pDescription, final int pWorth, final int pWeight) {
7     this.aName = pName;
8     this.aDescription = pDescription;
9     this.aWorth = pWorth;
10    this.aWeight = pWeight;
11 }
```

J'ai également ajouté trois accesseurs :

```
1 /**
2  * @return the weight of the item
3  */
4 public int getWeight() {
5     return this.aWeight;
6 }
7
```

On ajoute ensuite le poids à chaque déclaration d'item :

```

1 Item vLordBlessing = new Item("Lord\'s blessing", "holy relic", 10, 0);
2 Item vPriestRobe = new Item("Priest\'s robe", "holy relic", 5, 2);
3 Item vDeathScythe = new Item("Death\'s scythe", "unholy weapon to bring death upon mankind", -5, 6);
4 Item vIvoryCross = new Item("Ivory cross", "Holy cross from the holy crusades", 7, 1);
5 Item vHolyWater = new Item("Holy water", "Flask of water from Jerusalem", 2, 1);
6 Item vHellBible = new Item("Hell\'s bible", "Bible from the devil himself, contains evil but useful spells", -5, 3);
7 Item vBible = new Item("Bible", "Holy book from the Lord", 2, 1);
8 Item vHolyGrail = new Item("Holy grail", "Holy relic from the holy crusades", 10, 2);
9 Item vPurseOfCoins = new Item("Purse of coins", "The last 30 coins from the purse of Judas", -10, 1);
10 Item vSplinterOfCharon = new Item("Splinter of Charon", "Splinter of the boat of the underworld", 1, 3);
11 Item vViolentFire = new Item("Violent fire", "Fire from the violent, could be useful", -1, 1);
12 Item vBloodFlask = new Item("Blood flask", "Flask of blood from Phlegethon, the river of blood", -1, 2);
13 Item vMercyOfVirgil = new Item("Mercy of Virgil", "Mercy of the poet Virgil, the most respected pagan of hell", 5, 0);
14 Item vEyesOfTheGorgon = new Item("Eyes of the Gorgon", "Eyes of the Gorgon, the most feared monster of hell", -3, 3);
15 Item vHeadOfLucifer = new Item("Head of Lucifer", "Head of Lucifer, the most feared demon of hell", 0, 9);
16

```

On créer une méthode qui permet de constater si le Player courant peut take() l'item présenté :

```

1 public boolean canCarry(final int pItemWeight) {
2     if (this.getCurrentWeight() + pItemWeight <= this.aMaxWeight) {
3         return true;
4     } else {
5         return false;
6     }
7 }

```

Viens enfin la modification de take() dans GameEngine pour ne pas déranger Player inutilement :

```

1 if (!this.aPlayer.canCarry(this.aPlayer.getCurrentRoom().getItem(vItemName).getWeight())) {
2     this.aGui.println("You can't carry " + vItemName + " because it's too heavy. Drop some Items first.");
3     return;
4 }
5

```

On modifie finalement take() et drop() pour inclure le changement de poids :

```

1  /**
2   * Ajoute un item à l'inventaire du joueur
3   * @param pItemName
4   */
5  public void take(final String pItemName) {
6      this.aInventory.addItem(this.aCurrentRoom.getItem(pItemName));
7      this.aCurrentRoom.removeItem(pItemName);
8      this.aCurrentWeight += this.aInventory.getItem(pItemName).getWeight();
9  }
10
11 /**
12  * Retire un item de l'inventaire du joueur
13  * @param pItemName
14  */
15 public void drop(final String pItemName) {
16     this.aCurrentRoom.addItem(this.aInventory.getItem(pItemName).getName(), this.aInventory.getItem(pItemName));
17     this.aInventory.removeItem(pItemName);
18     this.aCurrentWeight -= this.aInventory.getItem(pItemName).getWeight();
19 }

```

7.33) Dans cet exercice j'ai décidé de refactor la pile de if else utilisé dans interpretCommand() afin de rendre l'ajout de commande plus simple :



```
1  String vCommandWord = vCommand.getCommandWord();
2  if ( vCommandWord.equals( "help" ) )
3      this.prinHelp();
4  else if ( vCommandWord.equals( "look" ) )
5      this.look();
6  else if ( vCommandWord.equals( "pray" ) )
7      this.pray();
8  else if ( vCommandWord.equals( "go" ) )
9      this.goRoom(vCommand);
10 else if ( vCommandWord.equals( "back" ) )
11     this.back();
12 else if ( vCommand.equals("take"))
13     this.take(vCommand);
14 else if ( vCommand.equals("drop"))
15     this.drop(vCommand);
16 else if ( vCommandWord.equals( "quit" ) ) {
17     if ( vCommand.hasSecondWord() )
18         this.aGui.println( "Quit what?" );
19     else
20         this.endGame();
21 } else {
22     this.aGui.println("Command not implemented yet");
23 }
```

Ancienne version ci-dessus, nouvelle version ci-dessous avec *items* implémenté :


```
1      switch (vCommandWord) {
2          case "help":
3              this.printHelp();
4              break;
5          case "look":
6              this.look();
7              break;
8          case "pray":
9              this.pray();
10             break;
11          case "go":
12              this.goRoom(vCommand);
13              break;
14          case "back":
15              this.back();
16              break;
17          case "take":
18              this.take(vCommand);
19              break;
20          case "drop":
21              this.drop(vCommand);
22              break;
23          case "quit":
24              if ( vCommand.hasSecondWord() )
25                  this.aGui.println( "Quit what?" );
26              else
27                  this.endGame();
28              break;
29          case "items":
30              this.printItems();
31              break;
32          default:
33              this.aGui.println("Command not implemented yet");
34              break;
35      }
36  }
```

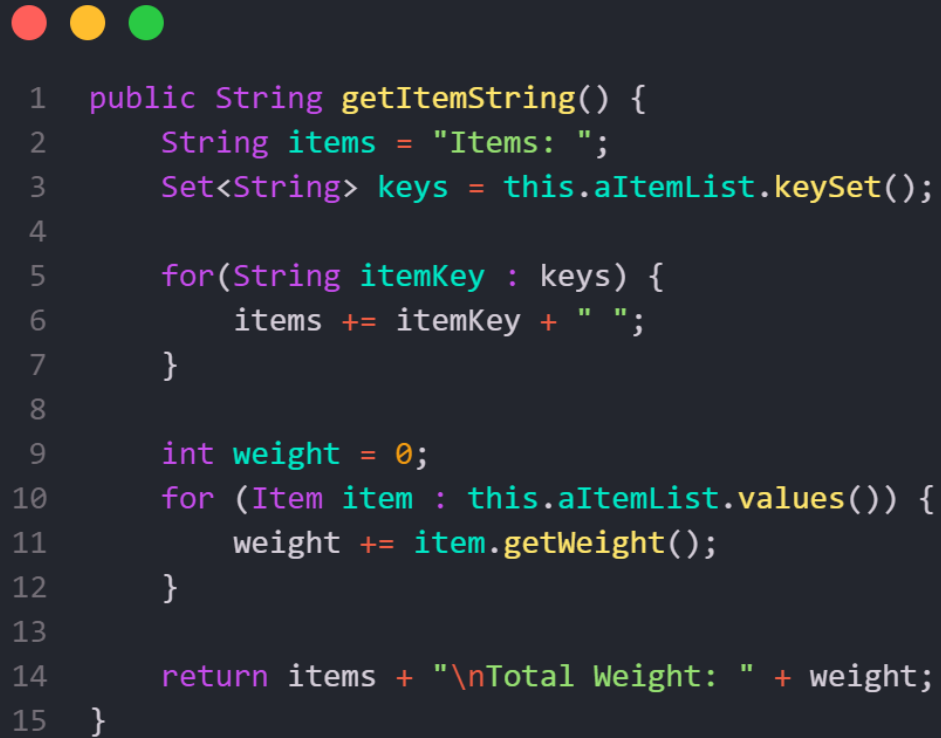
Viens maintenant la méthode printItems() qui communique avec Player et ItemList :



```
1  /**
2   * Print the items in the player's inventory
3   */
4  private void printItems() {
5      this.aGui.println(this.aPlayer.getInventoryString());
6  }
```



```
1  /**
2   * Retourne la String de l'inventaire du joueur
3   */
4  public String getInventoryString() {
5      return this.aInventory.getItemString();
6  }
```



```
1  public String getItemString() {
2      String items = "Items: ";
3      Set<String> keys = this.aItemList.keySet();
4
5      for(String itemKey : keys) {
6          items += itemKey + " ";
7      }
8
9      int weight = 0;
10     for (Item item : this.aItemList.values()) {
11         weight += item.getWeight();
12     }
13
14     return items + "\nTotal Weight: " + weight;
15 }
```

7.34) On étend la commande eat comme demandé dans l'énoncé (Je ne l'avais pas écrite je l'ai donc fait sur le tas)

```

1  /**
2   * Eat an item
3   * @param pCommand
4   */
5  private void eat(final Command pCommand) {
6      if (!pCommand.hasSecondWord()) {
7          this.aGui.println("Eat What ?");
8          return;
9      }
10
11     String vItemName = pCommand.getSecondWord();
12
13     if (!this.aPlayer.hasItem(vItemName)) {
14         this.aGui.println("You don't have " + vItemName + " in your inventory.");
15         return;
16     }
17
18     Item vItem = this.aPlayer.getCurrentRoom().getItem(vItemName);
19
20
21     this.aPlayer.eat(vItemName);
22     this.aGui.println("You eat " + vItemName + ".");
23 }

```

```

1  /**
2   * Mange un item de l'inventaire
3   * @param pItemName
4   */
5  public void eat(final String pItemName) {
6      this.aInventory.removeItem(pItemName);
7      this.aCurrentWeight -= this.aInventory.getItem(pItemName).getWeight();
8  }

```

On ajoute également comme demandé un *magic cookie*

```

1 Item vMagicCookie = new Item("Magic cookie", "A magic cookie that will give you double inventory capacity", 0, 0);
2
3 // Add Items to rooms
4 vLimbo.addItem(vLordBlessing.getName(), vLordBlessing);
5 vLimbo.addItem(vDeathScythe.getName(), vDeathScythe);
6
7 vLimboCitadel.addItem(vIvoryCross.getName(), vIvoryCross);
8
9 vLust.addItem(vPriestRobe.getName(), vPriestRobe);
10
11 vMarrakech.addItem(vHolyWater.getName(), vHolyWater);
12 vMarrakech.addItem(vMagicCookie.getName(), vMagicCookie);

```

On ajuste la méthode eat() dans GameEngine :

```

1 /**
2  * Eat an item
3  *
4  * @param pCommand
5  */
6 private void eat(final Command pCommand) {
7     if (!pCommand.hasSecondWord()) {
8         this.aGui.println("Eat What ?");
9         return;
10    }
11
12    String vItemName = pCommand.getSecondWord();
13
14    if (!this.aPlayer.hasItem(vItemName)) {
15        this.aGui.println("You don't have " + vItemName + " in your inventory.");
16        return;
17    }
18
19    // Item vItem = this.aPlayer.getCurrentRoom().getItem(vItemName);
20
21    this.aPlayer.eat(vItemName);
22    if (vItemName.equals("magiccookie")) {
23        this.aGui.println("You feel a strange power in you.");
24        this.aPlayer.setMaxWeight(this.aPlayer.getMaxWeight() * 2);
25    }
26    this.aGui.println("You eat " + vItemName + ".");
27 }
28

```

(j'ai entre temps changé le nom du magic cookie car les espaces ne font pas bon ménage avec le parser)

7.34.1)

Je n'ai pas encore mis à jour le fichier de test.

7.34.2) Comme au 7.26, on génère la javadoc avec les commandes données. Elle est donc accessible depuis la racine du projet.

III. Mode d'emploi (si nécessaire, instructions d'installation ou pour démarrer le jeu)

IV. Déclaration obligatoire anti-plagiat (*)