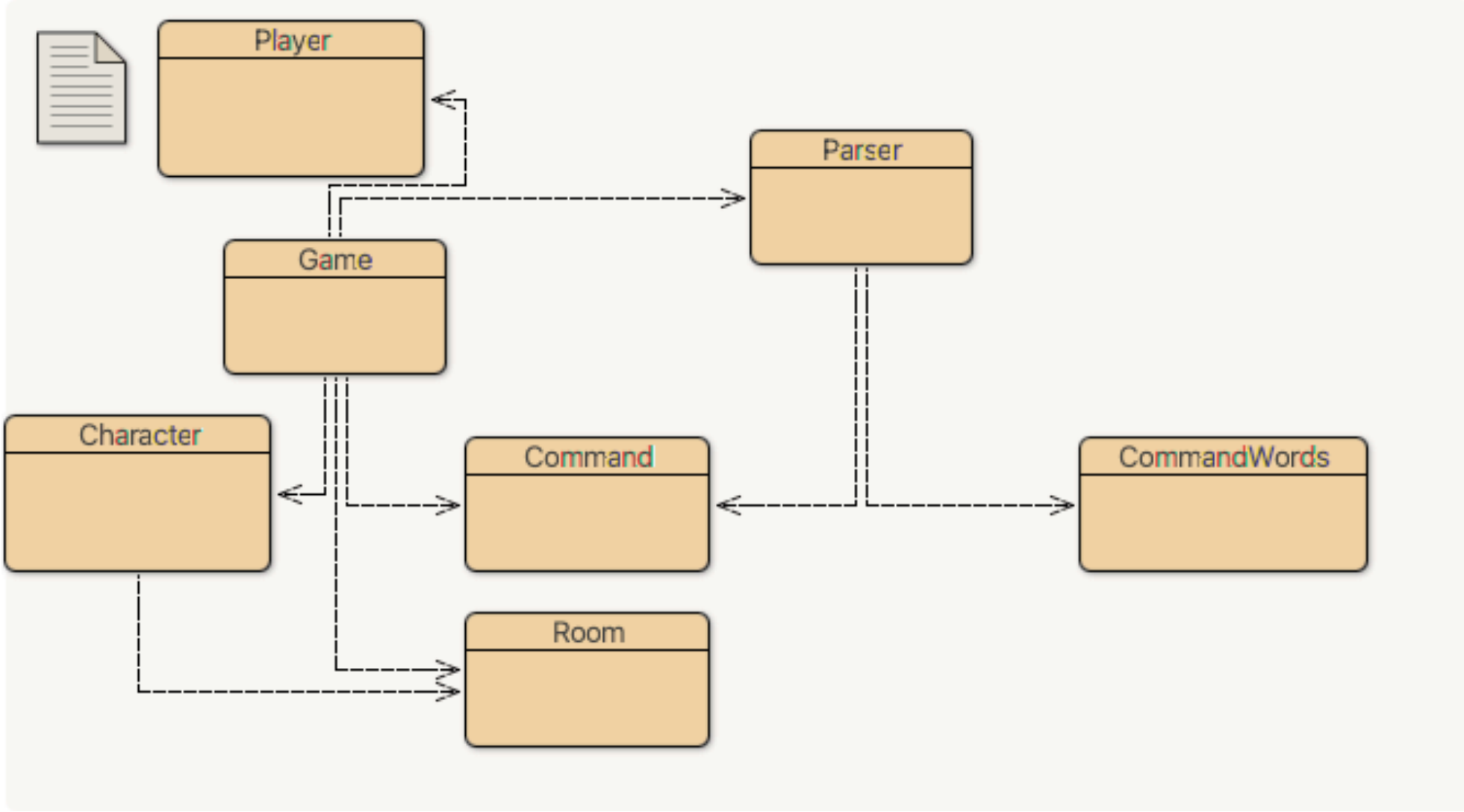
Player
Parser
Game
Character
Command
CommandWords
Room

```java
import java.util.HashMap;
import java.util.Set;

/**
 * Class Player - a player in an adventure game.
 *
 * This class is part of the "Magical House" application.
 * "Magical House" is a very simple, text based adventure game.
 *
 * A "Player" represents one player in the game. A player has a
 * bag in which it can carry items. Player can't carry all items
 * and can only carry items up to a certain weight.
 *
 * @author Zahra Amaan(K21011879).
 * @version 2021.12.03
 */
public class Player
{
    private int maxWeight = 50;// maximum weight that the player can carry in their bag.
    HashMap<String, Integer> bag; // the players bag, stores the name of the item and its corresponding weight
    Set<String> winningItems = Set.<String>of("sushi", "pizza", "tacos", "biryani", "paella");// items that the player needs to win
    Set<String> cantTakeItems = Set.<String>of("shoe", "button", "crayon");// items that the player can't take

    /**
     * Constructor for objects of class Player
     * initialises the bag and sets the maximum weight that the player can
     * carry in their bag
     */
    public Player()
    {
        bag = new HashMap<>();
        maxWeight = 30;
    }

    /**
     * Adds an item to the players bag.
     * @param item The item that the player wants to add to their bag.
     */
    public void addItemToBag (String item, int weight)
    {
        bag.put(item,weight);
    }

    /**
     * checks an item can be taken by the player. Returns true if the player
     * can take the item and returns false if the player can't take the item.
     * @param item The item that the player wants to take.
     */
    public boolean canTake(String item)
    {
```

```
51          boolean canTake = true;
52          if(cantTakeItems.contains(item)){
53              canTake = false;
54          }
55          return canTake;
56      }
57
58      /**
59       * Removes an item from the players bag.
60       * @param item The item that the player wants to remove from their bag.
61       */
62      public void returnItemFromBag (String item)
63      {
64          bag.remove(item);
65      }
66
67      /**
68       * Calculates the total weight of all the items in the players bag.
69       * @Return total The total weight of the players bag.
70       */
71      private int totalWeight()
72      {
73          int total = 0;
74          for(String item : bag.keySet())
75          {
76             total = total + bag.get(item);
77          }
78          return total;
79      }
80
81      /**
82       * Checks if the players bag contains an item.
83       * @Param item The item that you are looking for in the players bag.
84       * @Return bag.containsKey(item) True if the item is in the players bag.
85       * False if the item is not in the players bag.
86       */
87      public boolean bagContains (String item)
88      {
89          return bag.containsKey(item);
90      }
91
92      /**
93       * Gets the weight of an item.
94       * @Param item The item whose weight you want to get.
95       * @Return bag.get(item) The weight of the item.
96       */
97      public int getItemWeight(String item)
98      {
99          return bag.get(item);
100     }
101
102     /**
103      * Checks if the players bag if full. Returns true if it is and false
104      * if it is not.
```

```
105      * @Return full True if the total weight of the bag is greater than or
106      * equal to the maximum weight that the player can carry. False if the
107      * bag is not full.
108      */
109     public boolean bagFull(){
110         boolean full = false;
111         if (totalWeight() >= maxWeight)
112         {
113             full = true;
114         }
115         return full;
116     }
117 
118     /**
119      * Checks if the players has won the game, by checking if their bag is
120      * a winning bag. If it is then method returns true, else returns false.
121      * @Return won True if the has won the game. False if the player has
122      * not won the game.
123      */
124     public boolean hasWon()
125     {
126         boolean won = false;
127         if(isWinningBag()==true){
128             won=true;
129         }
130         else{
131             won=false;
132         }
133         return won;
134     }
135 
136     /**
137      * Generates a string of all the items in the bag.
138      * @Return bagString The string of all items in the players bag.
139     */
140     public String getBagString(){
141         String bagString = "Bag:";
142         for(String item : bag.keySet()){
143             bagString = bagString + " "+ item;
144         }
145         return bagString;
146     }
147 
148     /**
149      * Generates a string of all the items in the bag and the total weight
150      * of the bag.
151      * @Return bagString The string of all items in the players bag and the
152      * total weight of the bag.
153     */
154     public String getBagStringLong(){
155         String bagString = "Bag:";
156         for(String item : bag.keySet()){
157             bagString = bagString + " "+ item;
158         }
```

```
159         bagString = bagString + "\nTotal weight: " + totalWeight();
160         return bagString;
161     }
162 
163     /**
164      * Checks if the players bag contains all the winning items. If it
165      * does the method returns true, otherwise the method returns false.
166      * @Return winningBag True if the players bag contains all the winning
167      * items. False if the players bag does not contain all the winning items.
168      */
169     private boolean isWinningBag()
170     {
171         boolean winningBag = false;
172         Set bagItems = bag.keySet();
173         if(bagItems.containsAll(winningItems)){
174             winningBag = true;
175         }
176         return winningBag;
177     }
178 
179     /**
180      * Increases the maximum weight that the player can carry in their bag.
181      */
182     public void increaseMaxWeight(){
183         maxWeight = maxWeight + 20;
184     }
185 
186     /**
187      * Returns the maximum weight that the player can carry.
188      * @Return maxWeight The maximum weight that the player can carry.
189      */
190     public int getMaxWeight(){
191         return maxWeight;
192     }
193 
194 }
195 
```

```java
import java.util.Random;
import java.util.Stack;

/**
 *  This class is the main class of the "Magical House" application.
 *  "Magical House" is a very simple, text based adventure game.  Users
 *  can walk around a house.
 *
 *  To play this game, create an instance of this class and call the "play"
 *  method.
 *
 *  This main class creates and initialises all the others: it creates all
 *  rooms, creates the parser, crates the player, creates the characters
 *  and starts the game.  It also evaluates and executes the commands
 *  that the parser returns.
 *
 * @author  Michael Kölling, David J. Barnes and Zahra Amaan(K21011879).
 * @version 2016.02.29
 */

public class Game
{
    private Parser parser;
    private Room currentRoom;
    private Stack<Room> previousRoom = new Stack<>();
    private Room portalRoom;
    private Player player;
    private Room[] rooms;
    private Character dragon;
    private Character fairy;

    /**
     * Create the game, initialise its internal map and array of rooms.
     */
    public Game()
    {
        rooms = new Room[5];
        createRooms();
        parser = new Parser();
        player = new Player();
    }

    /**
     * Create all the rooms, link their exits together and add items and
characters to the rooms.
     */
    private void createRooms()
    {
        Room japan, italy, mexico, india, spain;

        // create the rooms

        portalRoom = new Room("in the portal.");

```

```
54          japan = new Room("in Japan");
55          italy = new Room("in Italy");
56          mexico = new Room("in Mexico");
57          india = new Room("in India");
58          spain= new Room("in Spain");
59
60          // add rooms to rooms array
61          rooms[0] = japan;
62          rooms[1] = italy;
63          rooms[2] = mexico;
64          rooms[3] = india;
65          rooms[4] = spain;
66
67          // initialise room exits
68          japan.setExit("east", italy);
69          japan.setExit("south", mexico);
70
71          italy.setExit("east", portalRoom);
72          italy.setExit("south", india);
73          italy.setExit("west", japan);
74
75          mexico.setExit("north", japan);
76          mexico.setExit("east", india);
77
78          india.setExit("north", italy);
79          india.setExit("east", spain);
80          india.setExit("west", mexico);
81
82          spain.setExit("north", portalRoom);
83          spain.setExit("west", india);
84
85          // initialise directions to all other rooms
86          japan.setDirections("Italy: go east.\nMexico: go south.\nIndia: go south,
go east.\nSpain: go south, go east, go east.\nPortal: go east, go east.");
87          italy.setDirections("Japan: go west.\nMexico: go south, go west.\nIndia:
go south.\nSpain: go south, go east.\nPortal: go east.");
88          mexico.setDirections("Japan: go north.\nItaly: go north, go east.\nIndia:
go east.\nSpain: go east, go east.\nPortal: go north, go east, go east.");
89          india.setDirections("Japan: go north, go west.\nItaly: go north.\nMexico:
go west.\nSpain: go east.\nPortal: go north, go east.");
90          spain.setDirections("Japan: go west, go west, go north.\nItaly:go west, go
north\nMexico: go west, go west.\nIndia:go west.\nPortal: go north.");
91
92          // add items and wieghts of items to the rooms
93          japan.addItem("sushi",10);
94          japan.addItem("pasta",10);
95          japan.addItem("kebab",10);
96
97          italy.addItem("pretzel",10);
98          italy.addItem("pizza",10);
99          italy.addItem("shoe",100);
100
101         mexico.addItem("button",100);
102         mexico.addItem("spaghetti",10);
```

```
103         mexico.addItem("tacos",10);
104 
105         india.addItem("biryani",10);
106         india.addItem("ramen",10);
107         india.addItem("burrito",10);
108 
109         spain.addItem("poutine",10);
110         spain.addItem("crayon",100);
111         spain.addItem("paella",10);
112 
113         //add a power to a random room
114         getRandomRoom().addItem("power",0);
115 
116         //add characters to rooms
117         fairy = new Character(japan);
118         dragon = new Character(spain);
119 
120         // set the clues for each room
121         fairy.addSpeech(japan, "This dish contains raw fish,\nwrapped in seaweed
or rice,\nit always tastes so nice.");
122         fairy.addSpeech(italy, "This is something that smells cheesy,\nbut it's
not a pair of socks,\nit is a food that is sliced,\nand delivered in a box.");
123         fairy.addSpeech(mexico, "This has a hard or soft shell,\nwith meat to
eat,\nyou can top it with lettuce,\nor top it with cheese.");
124         fairy.addSpeech(india, "This is a dish of meat and rice,\nand it conains a
lot of spice.");
125         fairy.addSpeech(spain, "With rice, seafood, veg and chicken,\nthis dish is
made for sharing,\nit is cooked and server all in one pan,\ntry and finish it if
you can.");
126 
127         // set the facts for each room
128         dragon.addSpeech(japan, "In Japan KFC is a traditional Christmas eve
meal.");
129         dragon.addSpeech(italy, "In Italy salad is eaten after the meal (for
dessert).");
130         dragon.addSpeech(mexico, "The Caesar Salad was invented in Mexico.");
131         dragon.addSpeech(india, "The Indian cuisine is the oldest cuisine in human
history.");
132         dragon.addSpeech(spain, "In Spain there is a New Years custom where you
eat 13 grapes.");
133 
134         currentRoom = japan;  // start game in japan
135     }
136 
137     /**
138      *  Main play routine. Loops until end of play.
139      */
140     public void play()
141     {
142         printWelcome();
143 
144         // Enter the main command loop.  Here we repeatedly read commands and
145         // execute them until the game is over.
146 
147         boolean finished = false;
```

```
149             Command command = parser.getCommand();
150             finished = processCommand(command);
151         }
152         System.out.println("Thank you for playing.  Good bye.");
153     }
154 
155     /**
156      * Print out the opening message for the player.
157      */
158     private void printWelcome()
159     {
160         System.out.println();
161         System.out.println("Welcome to the Magical House!");
162         System.out.println("In this house of magic each room is a different
country.");
163         System.out.println("Your mission is to go home.");
164         System.out.println("In order to go home you must go to each country and
take their traditional food.");
165         System.out.println("But the total weight you can carry is " +
player.getMaxWeight());
166         System.out.println("You must find and gain power item to be able to carry
more weight.");
167         System.out.println("On your mission you will be accompanied by a fairy,
who you can ask for clues.");
168         System.out.println("And you may also run into a dragon, who you can ask
for facts about the country you are in.");
169         System.out.println("Type 'help' if you need help.");
170         System.out.println();
171         System.out.println(currentRoom.getLongDescription());
172         System.out.println(player.getBagString());
173         isFairy(currentRoom);// prints a message if there is a fairy in the first
room
174     }
175 
176     /**
177      * Given a command, process (that is: execute) the command.
178      * @param command The command to be processed.
179      * @return true If the command ends the game, false otherwise.
180      */
181     private boolean processCommand(Command command)
182     {
183         boolean wantToQuit = false;
184         boolean hasWon = false;
185 
186         if(command.isUnknown()) {
187             System.out.println("I don't know what you mean...");
188             return false;
189         }
190 
191         String commandWord = command.getCommandWord();
192         if (commandWord.equals("help")) {
193             printHelp();
194         }
195         else if (commandWord.equals("go")) {
196             if(command.hasSecondWord()&&command.getSecondWord().equals("home"))
```

```
197             {
198                 hasWon=goHome();//signal that player has won. method returns a boolean.
199             }
200             else{
201                 goRoom(command);
202             }
203         }
204         else if (commandWord.equals("quit")) {
205             wantToQuit = quit(command);
206         }
207         else if (commandWord.equals("back")) {
208             goBack();
209         }
210         else if (commandWord.equals("take")) {
211             takeItem(command);
212         }
213         else if (commandWord.equals("return")) {
214             returnItem(command);
215         }
216         else if (commandWord.equals("ask")) {
217             if(fairy.inRoom(currentRoom)||dragon.inRoom(currentRoom)){
218                 ask(command);//user can only use ask command if there is a character in the room.
219             }
220             else{
221                 System.out.println("There is no one to ask in this room");
222             }
223         }
224         else if (commandWord.equals("directions")) {
225             printDirections(currentRoom);
226         }
227         else if (commandWord.equals("map")) {
228             map();
229         }
230         else if (commandWord.equals("bag")) {
231             bag();
232         }
233          else if (commandWord.equals("gain")) {
234             if(!currentRoom.containsItem("power")){
235                 System.out.println("There is nothing to gain in this room");
236             }
237             else{
238                 gain(command);//user can onlt use the gain command if there is something to gain in the room
239             }
240         }
241 
242         // else command not recognised.
243 
244         return (wantToQuit||hasWon); // returns signal to quit if the user wants to quit or if the player has won the game.
245     }
246 
```

```
247     // implementations of user commands:
248
249     /**
250      * Print out some help information.
251      * Here we print some stupid, cryptic message and a list of the
252      * command words.
253      */
254     private void printHelp()
255     {
256         System.out.println("You are lost. You are alone. You wander");
257         System.out.println("around the Magical House.");
258         System.out.println();
259         System.out.println("Your command words are:");
260         System.out.println(parser.showCommands());//diplays all the command words
261     }
262
263     /**
264      * Try to in to one direction. If there is an exit, enter the new
265      * room, otherwise print an error message. If the next room is the
266      * portalRoom then execte the portal method.
267      */
268     private void goRoom(Command command)
269     {
270         if(!command.hasSecondWord()) {
271             // if there is no second word, we don't know where to go...
272             System.out.println("Go where?");
273             return;
274         }
275
276         String direction = command.getSecondWord();
277
278         // Try to leave current room.
279         Room nextRoom = currentRoom.getExit(direction);
280
281         if(nextRoom == portalRoom)
282         {
283             portal();
284             return;
285         }
286
287
288         if (nextRoom == null) {
289             System.out.println("There is no door!");
290         }
291         else {
292             previousRoom.push(currentRoom);//pushes the previous room onto the
    stack
293             currentRoom = nextRoom;
294             changeFairyRoom();
295             changeDragonRoom();
296
297             newRoomText();
298         }
299     }
300
```

```
301     /**
302      * Checks if the user has won the game. If they have a message is
303      * printed out and the game terminates. If they have not won a message is
304      * printed and the game continues.
305      * @return won True if they have won the game, False otherwise.
306      */
307     private boolean goHome()
308     {
309         boolean won = false;
310         if (player.hasWon()){
311             System.out.println("Congratualations you have completed your mission
and won the game.");
312             won = true;
313         }
314         else
315         {
316             System.out.println("You have not completed the mission yet, keep
trying.");
317         }
318         return won;
319     }
320 
321     /**
322      * "Quit" was entered. Check the rest of the command to see
323      * whether we really quit the game.
324      * @return true, if this command quits the game, false otherwise.
325      */
326     private boolean quit(Command command)
327     {
328         if(command.hasSecondWord()) {
329             System.out.println("Quit what?");
330             return false;
331         }
332         else {
333             return true;  // signal that we want to quit
334         }
335     }
336 
337     /**
338      * Takes the player to the previous room that they were in.
339      */
340     private void goBack()
341     {
342         if(!previousRoom.empty()){
343             currentRoom = previousRoom.pop();//pops previous room from top of
stack
344             System.out.println(currentRoom.getLongDescription());
345         }
346         else{
347             System.out.println("You have gone back to the first room you were in,
you can't go back any more.");
348         }
349     }
350 
351     /**
```

```
352      * Takes an item from the current room of the player
353      * and puts it in the players bag.
354      * The player can't take the item if the item is not in the room,
355      * the item is not food, the item is power, or the players bag is full.
356      */
357     private void takeItem(Command command)
358     {
359         if(!command.hasSecondWord()) {
360             // if the command doesn't have a second word, we don't know what the player wants to take.
361             System.out.println("Take what?");
362             return;
363         }
364         String item = command.getSecondWord();
365         if(!currentRoom.containsItem(item))
366         {
367             System.out.println("You cannot take this item because it is not in the room.");
368         }
369         else if(!player.canTake(item)){
370             System.out.println("You cannot take this item because it is not food.");
371         }
372         else if(player.bagFull()){
373             System.out.println("You cannot take this item because your bag is full.");
374         }
375         else if (item.equals("power")){
376             System.out.println("You can't take power, you must gain power.");
377         }
378         else{
379             player.addItemToBag(item,currentRoom.getItemWeight(item));
380             currentRoom.removeItem(item);
381             if(player.bagFull()){
382                 System.out.println("Your bag is now full. Gain power to be able to take more items.");
383                 System.out.println("If you have already gained power, you cannot take any more items unless you return some first.");
384                 System.out.println("If you think you have completed the mission, go home.");
385             }
386         }
387     }
388 
389     /**
390      *Takes an item from the players bag and returns it to the current room of
391      *the player.
392      *The player can't return an item if it is not in their bag.
393      */
394     private void returnItem(Command command)
395     {
396         if(!command.hasSecondWord()) {
397             // if the command doesn't have a second word, we don't know what the player wants to return.
398             System.out.println("Return what?");
```

```
400         }
401         String item = command.getSecondWord();
402         if(!player.bagContains(item))
403         {
404             System.out.println("You cannot return this item because it is not in your bag.");
405         }
406         else
407         {
408         currentRoom.addItem(item, player.getItemWeight(item));
409         player.returnItemFromBag(item);
410         }
411     }
412 
413     /**
414      * Prints out a characters speech if the user inputs a valis command.
415      * "ask fairy clue" prints out the fairys speech for the current room.
416      * "ask dragon fact" prints out the dragons speech for the current room.
417      */
418     private void ask(Command command)
419     {
420         if (!command.hasSecondWord()){
421             // if there is no second word, we don't know who to ask.
422             System.out.println("Ask who?");
423             return;
424         }
425 
426         String secondWord = command.getSecondWord();
427 
428         // if the second word is not valid
429         if(!secondWord.equals("fairy")&& !secondWord.equals("dragon")) {
430             if (fairy.inRoom(currentRoom)&&dragon.inRoom(currentRoom)){
431                 System.out.println("You must ask the fairy or the dragon.");
432                 return;
433             }
434             else if(fairy.inRoom(currentRoom)){
435                 System.out.println("You must ask the fairy.");
436                 return;
437             }
438             else if(dragon.inRoom(currentRoom)){
439                 System.out.println("You must ask the dragon.");
440                 return;
441             }
442             }
443 
444         // if there is no third word, we don't know what to ask the characters.
445         if (!command.hasThirdWord()){
446                 if(fairy.inRoom(currentRoom)&&secondWord.equals("fairy"))
447                 {
448                     System.out.println("Ask fairy what? ");
449                     return;
450                 }
451                 else if (!fairy.inRoom(currentRoom)&&secondWord.equals("fairy")){
452                     System.out.println("There is no fairy in this room.");
```

```
453                     return;
454                 }
455                 if(dragon.inRoom(currentRoom)&&secondWord.equals("dragon"))
456                 {
457                     System.out.println("Ask dragon what? ");
458                     return;
459                 }
460                 else if(!dragon.inRoom(currentRoom)&&secondWord.equals("dragon")){
461                     System.out.println("There is no dragon in this room. ");
462                     return;
463                 }
464             }
465 
466         String thirdWord = command.getThirdWord();
467 
468         if(secondWord.equals("fairy")&&!thirdWord.equals("clue"))
469         {
470                 System.out.println("You must ask the fairy for a clue.");
471                 return;
472         }
473 
474         if(secondWord.equals("dragon")&&!thirdWord.equals("fact"))
475         {
476                 System.out.println("You must ask the dragon for a fact.");
477                 return;
478         }
479 
480         // valid commands:
481 
482         if(secondWord.equals("fairy")&&thirdWord.equals("clue"))
483         {
484             if (fairy.inRoom(currentRoom))
485             {
486                 System.out.println(fairy.getSpeech(currentRoom));
487             }
488         }
489 
490         if(secondWord.equals("dragon")&&thirdWord.equals("fact"))
491         {
492             if (dragon.inRoom(currentRoom))
493             {
494                 System.out.println(dragon.getSpeech(currentRoom));
495             }
496         }
497 
498     }
499 
500     /**
501      *Takes a room as a parameter and prints out the directions of how to
502      *get to all the other rooms, from that room.
503      *@param room The room that the player is in.
504      */
505     private void printDirections(Room room)
506     {
```

```
        System.out.println(room.getDirections());
    }

    /**
     * Prints strings that resemble a map of the game.
     */
    private void map(){
    System.out.println(" _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ ");
    System.out.println("|                  |                  |                  |");
    System.out.println("|      Japan       |       Italy      |       Portal     |");
    System.out.println("|_ _ _ _ _ _ _ _ _ |_ _ _ _ _ _ _ _ _ |_ _ _ _ _ _ _ _ _ |");
    System.out.println("|                  |                  |                  |");
    System.out.println("|      Mexico      |       India      |       Spain      |");
    System.out.println("|_ _ _ _ _ _ _ _ _ |_ _ _ _ _ _ _ _ _ |_ _ _ _ _ _ _ _ _ |");
    }

    /**
     * Prints out all the items in the players bag and the total weight of all the items.
     */
    private void bag(){
        System.out.println(player.getBagStringLong());
    }

    /**
     * Removes an item from the current room and the player gains the item.
     * "gain power" the power item is removed from the current room and the total weight that the player can
     * carry increases.
     */
    private void gain(Command command){
        if(!command.hasSecondWord()){
            // if there is no second word, we don't what to gain.
            System.out.println("Gain what? ");
            return;
        }

        String powerUp = command.getSecondWord();

        if(!powerUp.equals("power") ){
            System.out.println("You must gain the power."); // the player must gain a valid item.
        }
        else{
            currentRoom.removeItem(powerUp);
            player.increaseMaxWeight();
            System.out.println("You have gained power and now you can carry a maximum weight of " + player.getMaxWeight() + ".");
        }
```

```
557      * Generates a random room by generating a random index for the array
558      * of rooms.
559      * @return a random room from the array of rooms.
560      */
561     private Room getRandomRoom()
562     {
563         Random r = new Random();
564         int n = r.nextInt(rooms.length-1);//gets a random number between 0 and one less than the number of rooms in the rooms array.
565         return rooms[n];// so n can represent any element in the array
566     }
567 
568     /**
569      * Takes the player to a random room
570      * and changes the room that the fairy and dragon are in.
571      */
572     private void portal()
573     {
574         portalText();
575 
576         Room randomRoom = getRandomRoom();
577         previousRoom.push(currentRoom);
578         currentRoom = randomRoom;
579 
580         changeFairyRoom();
581 
582         changeDragonRoom();
583 
584         newRoomText();
585     }
586 
587     /**
588      * Prints a message that tells the player about the portal.
589      */
590     private void portalText(){
591         System.out.println();
592         System.out.println("You are " + portalRoom.getShortDescription());
593         System.out.println("You will be transported to a random room.");
594         System.out.println();
595     }
596 
597     /**
598      * Prints a message that tells the player about the new room.
599      */
600     private void newRoomText(){
601         System.out.println(currentRoom.getLongDescription());
602         System.out.println(player.getBagString());
603         isFairy(currentRoom);
604         isDragon(currentRoom);
605     }
606 
607     /**
608      * Changes the room that the fairy is in to the current room of the player.
609      */
```

```
610     private void changeFairyRoom(){
611         fairy.changeRoom(currentRoom);
612     }
613 
614     /**
615      * Randomly changes the room that the dragon is in.
616      */
617     private void changeDragonRoom(){
618         Room randomRoom = getRandomRoom();
619         dragon.changeRoom(randomRoom);
620     }
621 
622     /**
623      * Checks if there is a fairy in the currentRoom that the player is in.
624      * A statement is printed out to the user if there is a fairy in the current
    room
625      * otherwise nothing happens.
626      * @Param room The room that you check if the fairy is in.
627      */
628     private void isFairy(Room room){
629         if(fairy.inRoom(currentRoom)){
630                 System.out.println("There is a fairy in this room.");
631             }
632     }
633 
634     /**
635      * Checks if there is a dragon in the currentRoom that the player is in.
636      * A statement is printed out to the user if there is a dragon in the current
    room
637      * otherwise nothing happens.
638      * @Param room The room that you check if the dragon is in.
639      */
640     private void isDragon(Room room){
641         if(dragon.inRoom(currentRoom)){
642                 System.out.println("There is a dragon in this room.");
643             }
644     }
645 }
646 
```

```java
import java.util.HashMap;

/**
 * Class Character - a Character in an adventure game.
 *
 * This class is part of the "Magical House" application.
 * "Magical House" is a very simple, text based adventure game.
 *
 * A "Character" represents one character in the game.
 * A character has speeches. Each speech is for a particular room.
 *
 * @author Zahra Amaan(K21011879).
 * @version 2021.12.03
 */
public class Character
{
    private Room currentRoom;
    private HashMap<Room, String> speeches;

    /**
     * Constructor for objects of class Character
     * Initialises the current room of the character.
     * @Param room The room you want to initialise the characters current room to.
     */
    public Character(Room room)
    {
        currentRoom = room;
        speeches = new HashMap<>();
    }

     /**
     * Adds the characters speech for a room.
     * @param  room The room you want to add the characters speech for.
     */
    public void addSpeech(Room room, String speech)
    {
        speeches.put(room,speech);//key is the room and the value is the speech
    }

     /**
     * Generates a string of the characters speech for a room.
     * @param  room The room you want to get the characters speech for.
     * @Return speech The characters speech for a room.
     */
    public String getSpeech(Room room){
        String speech = "";
        speech = speech + speeches.get(room);//returns the value associated with the key
        return speech;
    }

     /**
     * Checks if a character is in a room.
     * @param  room The room you want to check the character is is.
```

```
     * @Return (room==currentRoom) True if a character is in the
     * room provided. False if a character is not in the room
     */
    public boolean inRoom(Room room){
        return (room==currentRoom);
    }

    /**
     * Changes the current room of the character.
     * @param  room The room to change the characters room to.
     */
    public void changeRoom(Room room){
        currentRoom = room;
    }
}

```

```java
/**
 * This class is part of the "Magical House" application.
 * "Magical House" is a very simple, text based adventure game.
 *
 * This class holds information about a command that was issued by the user.
 * A command currently consists of three strings: a command word, a second
 * word and a third word (for example, if the command was "ask fairy clue", then the three strings
 * obviously are "ask", "fairy" and "clue".).
 *
 * The way this is used is: Commands are already checked for being valid
 * command words. If the user entered an invalid command (a word that is not
 * known) then the command word is <null>.
 *
 * If the command had only one word, then the second word and third word are <null>.
 *
 * @author  Michael Kölling, David J. Barnes and Zahra Amaan(K21011879).
 * @version 2016.02.29
 */

public class Command
{
    private String commandWord;
    private String secondWord;
    private String thirdWord;

    /**
     * Create a command object. First, second and third word must be supplied, but
     * any one (or all) of them can be null.
     * @param firstWord The first word of the command. Null if the command
     *                  was not recognised.
     * @param secondWord The second word of the command.
     * @param thirdWord The third word of the command.
     */
    public Command(String firstWord, String secondWord,String thirdWord)
    {
        commandWord = firstWord;
        this.secondWord = secondWord;
        this.thirdWord = thirdWord;
    }

    /**
     * Return the command word (the first word) of this command. If the
     * command was not understood, the result is null.
     * @return The command word.
     */
    public String getCommandWord()
    {
        return commandWord;
    }

    /**
     * @return The second word of this command. Returns null if there was no
     * second word
```

```
     */
    public String getSecondWord()
    {
        return secondWord;
    }

    /**
     * @return The third word of this command. Returns null if there was no
     * third word.
     */
    public String getThirdWord()
    {
        return thirdWord;
    }

    /**
     * @return true if this command was not understood.
     */
    public boolean isUnknown()
    {
        return (commandWord == null);
    }

    /**
     * @return true if the command has a second word.
     */
    public boolean hasSecondWord()
    {
        return (secondWord != null);
    }

    /**
     * @return true if the command has a third word.
     */
    public boolean hasThirdWord()
    {
        return (thirdWord != null);
    }
}
```

```java
/**
 * This class is part of the "Magical House" application.
 * "Magical House" is a very simple, text based adventure game.
 *
 * This class holds an enumeration of all command words known to the game.
 * It is used to recognise commands as they are typed in.
 *
 * @author  Michael Kölling, David J. Barnes and Zahra Amaan(K21011879).
 * @version 2016.02.29
 */

public class CommandWords
{
    // a constant array that holds all valid command words
    private static final String[] validCommands = {
        "go", "quit", "help", "back", "take", "return", "ask", "directions", "map", "bag", "gain"
    };

    /**
     * Constructor - initialise the command words.
     */
    public CommandWords()
    {
        // nothing to do at the moment...
    }

    /**
     * Check whether a given String is a valid command word.
     * @return true if it is, false if it isn't.
     */
    public boolean isCommand(String aString)
    {
        for(int i = 0; i < validCommands.length; i++) {
            if(validCommands[i].equals(aString))
                return true;
        }
        // if we get here, the string was not found in the commands
        return false;
    }

    /**
     * @Return validCommands The string of all valid command words.
     */
    public String getAll()
    {
        String validCommandWords = "";
        for(String command: validCommands) {
            validCommandWords = validCommandWords + " " + command;
        }
        return validCommandWords;
    }
}

```

```java
import java.util.Set;
import java.util.HashMap;
import java.util.HashSet;

/**
 * Class Room - a room in an adventure game.
 *
 * This class is part of the "Magical House" application.
 * "Magical House" is a very simple, text based adventure game.
 *
 * A "Room" represents one location in the scenery of the game.  It is
 * connected to other rooms via exits.  For each existing exit, the room
 * stores a reference to the neighbouring room. A room contains items. For
 * each item, the room stores the name of the item and the weight
 * of the item.
 *
 * @author  Michael Kölling, David J. Barnes and Zahra Amaan(K21011879).
 * @version 2016.02.29
 */

public class Room
{
    private String description;
    private HashMap<String, Room> exits; // stores exits of this room.
    private HashMap<String, Integer> items; // stores the items in this room.
    private String directions; // the directions to all other rooms from this room.

    /**
     * Create a room described "description". Initially, it has
     * no exits. "description" is something like "a kitchen" or
     * "an open court yard".
     * @param description The room's description.
     */
    public Room(String description)
    {
        this.description = description;
        exits = new HashMap<>();
        items = new HashMap<>();
    }

    /**
     * Define an exit from this room.
     * @param direction The direction of the exit.
     * @param neighbor  The room to which the exit leads.
     */
    public void setExit(String direction, Room neighbor)
    {
        exits.put(direction, neighbor);
    }

    /**
     * Return the room that is reached if we go from this room in direction
     * "direction". If there is no room in that direction, return null.
     * @param direction The exit's direction.
```

```
55      * @return The room in the given direction.
56      */
57     public Room getExit(String direction)
58     {
59         return exits.get(direction);
60     }
61 
62     /**
63      * Adds an item to this room.
64      * @param item The name of the item to add.
65      * @param weight The weight of the item.
66      */
67     public void addItem(String item, int weight)
68     {
69         items.put(item, weight);
70     }
71 
72     /**
73      * Removes an item from this room.
74      * @param item The name of the item to remove.
75      */
76     public void removeItem(String item)
77     {
78         items.remove(item);
79     }
80 
81     /**
82      * Checks if the room contains an item.
83      * @Param item The item that you are looking for in the room.
84      * @Return items.containsKey(item) True if the item is in the room.
85      * False if the item is not in the room.
86      */
87     public boolean containsItem(String item)
88     {
89         return items.containsKey(item);
90     }
91 
92     /**
93      * Gets the weight of an item.
94      * @Param item The item whose weight you want to get.
95      * @Return items.get(item) The weight of the item.
96      */
97     public int getItemWeight(String item)
98     {
99         return items.get(item);
100    }
101 
102    /**
103     * @return The short description of the room
104     * (the one that was defined in the constructor).
105     */
106    public String getShortDescription()
107    {
108        return description;
```

```
109     }
110
111     /**
112      * Return a description of the room in the form:
113      *     You are in the kitchen.
114      *     Exits: north west
115      * @return A long description of this room
116      */
117     public String getLongDescription()
118     {
119         return "You are " + description + ".\n" + getExitString() + ".\n" +
    getItemsString();
120     }
121
122     /**
123      * Return a string describing the room's exits, for example
124      * "Exits: north west".
125      * @return Details of the room's exits.
126      */
127     private String getExitString()
128     {
129         String returnString = "Exits:";
130         Set<String> keys = exits.keySet();
131         for(String exit : keys) {
132             returnString += " " + exit;
133         }
134         return returnString;
135     }
136
137     /**
138      * Return a string describing the room's items, for example
139      * "Items: pizza pretzel shoe".
140      * @return Details of the room's items.
141      */
142     private String getItemsString()
143     {
144         String itemsString = "Items:";
145         for(String item : items.keySet())
146         {
147             itemsString = itemsString + " " + item;
148         }
149         return itemsString;
150     }
151
152     /**
153      * Define the directions from the room to all other rooms.
154      * @param directions The directions from the room to all other rooms.
155      */
156     public void setDirections(String directions){
157         this.directions = directions;
158     }
159
160     /**
161      * Return a string describing the directions to all other rooms from
```

```
     * the room.
     * @return Details of the directions to get to all other rooms from
     * this room.
     */
    public String getDirections(){
        return directions;
    }
}


```

```java
import java.util.Scanner;

/**
 * This class is part of the "Magical House" application.
 * "Magical House" is a very simple, text based adventure game.
 *
 * This parser reads user input and tries to interpret it as an "Adventure"
 * command. Every time it is called it reads a line from the terminal and
 * tries to interpret the line as a three word command. It returns the command
 * as an object of class Command.
 *
 * The parser has a set of known command words. It checks user input against
 * the known commands, and if the input is not one of the known commands, it
 * returns a command object that is marked as an unknown command.
 *
 * @author  Michael Kölling, David J. Barnes and Zahra Amaan(K21011879).
 * @version 2016.02.29
 */
public class Parser
{
    private CommandWords commands;  // holds all valid command words
    private Scanner reader;         // source of command input

    /**
     * Create a parser to read from the terminal window.
     */
    public Parser()
    {
        commands = new CommandWords();
        reader = new Scanner(System.in);
    }

    /**
     * @return The next command from the user.
     */
    public Command getCommand()
    {
        String inputLine;   // will hold the full input line
        String word1 = null;
        String word2 = null;
        String word3 = null;

        System.out.print("> ");     // print prompt

        inputLine = reader.nextLine();

        // Find up to two words on the line.
        Scanner tokenizer = new Scanner(inputLine);
        if(tokenizer.hasNext()) {
            word1 = tokenizer.next();      // get first word
            if(tokenizer.hasNext()) {
                word2 = tokenizer.next();      // get second word
                // note: we just ignore the rest of the input line.
                if(tokenizer.hasNext()) {
```

```
                word3 = tokenizer.next();      // get third word
            }
            }
        }

        // Now check whether this word is known. If so, create a command
        // with it. If not, create a "null" command (for unknown command).
        if(commands.isCommand(word1)) {
            return new Command(word1, word2, word3);
        }
        else {
            return new Command(null, word2, word3);
        }
    }

    /**
     * @Return commands.getAll() The string of all valid command words.
     */
    public String showCommands()
    {
        return commands.getAll();
    }
}

```

```
Project: zuul-better
Authors: Michael Kölling and David J. Barnes

This project is part of the material for the book

   Objects First with Java - A Practical Introduction using BlueJ
   Sixth edition
   David J. Barnes and Michael Kölling
   Pearson Education, 2016

This project is a simple framework for an adventure game. In this version,
it has a few rooms and the ability for a player to walk between these rooms.
That's all.

To start this application, create an instance of class "Game" and call its
"play" method.

This project was written as the starting point of a small Java project.

The goal is to extend the game:

  - add items to rooms (items may have weight)
  - add multiple players
  - add commands (pick, drop, examine, read, ...)
  - (anything you can think of, really...)

Read chapter 8 of the book to get a detailed description of the project.

```