Assignment 2 report

"The Peacemaker"
Version: 1.1

King‘s College London
Faculty of Natural & Mathematical Sciences
Department of Informatics
Module: 4CCS1PPA
Vakaris Paulavičius (Student number: 20062023)

2020-11-30

Content:

Introduction

"The peacemaker" is a simple text-based adventure game. Here a player travels throughout a fictitious map win a goal to defeat the main boss and bring peace to the village as well as a chest full of coins.

In this game, a player can visit four islands (excluding the start point – main land), which all have their magnificent places, mobs, and items. A player can collect those items, leave them in any room, kill mobs, and collect their trophies.

Words in blue represent terms, methods, variables, commands and places used in the implementation.

Things to have in mind

> To increment health, a player must eat food items that are in his backpack.
> Before hitting a mob, a player should equip a weapon or else, he won‘t do any damage.
> A player cannot leave a place if there is at least one mob that is alive.
> To win the game, a player must go out from the boss room with a chest in his backpack. The chest is a trophy that is dropped by the main boss after his death.
> Every mob does a different amount of damage from their damage range (minDamage – maxDamage).
> Every mob has a different chance to drop a trophy (boss mob drops a chest every time).
> The teleport can take a player to any room, except the boss room, the main land, and the place which is considered “the last stop“.

At the beginning of the game, the user is asked to enter the preferred gender. According to the choice, a random female or male name is generated (there are two arrays with pre-defined fictitious names).

There is a primitive drawn map below which will help a player imagine the game map a little more easily.

Base tasks

- Some rooms were already created, so I just added my own rooms following the example.
- This was already implemented.
- I added 3 HashMaps (weapons, foodList, trophies) and an ArrayList (mobs) to the Room class. Each of the collections hold items that are currently in that room. Initial items are added with the creation of the game map. Later, a player can either pick up items that are in his currentRoom, or leave redundant items from his backpack in that room. A player cannot pick up mobs, he can only kill them.
- To implement this, I created a new class called Backpack. Its object represents a player‘s backpack. Every player has a backpack. Each backpack has a limited size (which is determined when the player object is created). A player can add items only if there is enough space. A backpack consists of three HashMaps (weapons, trophies, foodList). I tried making a backpack with a single HasMap – items (Item class for all things), but came up with a low

cohesion problem - different items have different attributes (weapons: damage, rarity, food products: health increment value).

- If a player goes out from the boss room and has a chest in his backpack – he wins. Every time a player enters a new location, this case is considered. If it turns out that a player is in the endRoom and has the necessary trophy - goal, the main game loop is stopped and the winning message is printed out to the terminal.
- I implemented this by adding additional 'else ifs' to the goRoom method in the Game class, which check for the user input. If the command is go back, then the current and previous locations are considered and appropriate actions are carried out. I am quite sure that this is not a good way to implement this functionality.
- I added new command words to the validCommands array and implemented their functionality in the Game class. New commands that I added: hit, eat, look, take, drop, info, list, stats, equip, about, show. The purpose of commands and how to use them are described in the README.txt file or one page below.

Challenge tasks

- I added characters to the game by making a separate class called Mob which represents a single mob. Mobs are added to different places when the map is generated. Unfortunately, mobs in my game cannot travel to other places. However, mobs can fight the player and drop trophies if killed.
- I made some changes in the classes Command and Parser so those classes would be able to recognize the third word and do some actions with it.
- I created a new room called teleport and added an additional else if to the goRoom method to execute the teleportToRoom method if the next room is the teleport.
- * I made a character creation at the start of the game. Before the main game loop, a user is asked to enter a gender according to which a player object is created. This promotes gender equality and gives more choice to the user.
- * I implemented the functionality of killing mobs. Whenever a player enters a room which has at least one mob in it, he cannot leave until all the mobs are defeated. This prevents a player from cheating and completing the game without putting any effort. I also added specific pattern for fighting as well as a funny implementation for a situation when a player tries to hit a mob without a weapon in his hand.

Code quality

- Coupling:
  - In my opinion, one example of loose coupling (which I implemented) is that all valid commands are no longer printed to the terminal when the showAll method in the CommandWords class is invoked. Instead, a string with all valid commands is returned via getAll method. This allows us to show these commands in a GUI for example instead of printing them out into the terminal.
  - Another example of loose coupling is how all items are listed when the look command is called. The system automatically follows all the existing items and returns a string with only currently available items in their "most of the time" correct singular and plural forms (look at the Bugs and problems section).

- Responibility-driven design:
  - I think that separate classes for different kinds of items, a player, a backpack and a map is an instance of responsibility-driven design.

- Maintainability:
  - One example of good maintainability is the use of lists and maps to store objects instead of creating separate objects and using them one by one.
  - Another example is a discrete class for printing text - TextCreator. It helps with readability and code elegance. Besides, it helps not to get lost in the code while trying to find a place where a particular text is being printed out.
  - However, TextCreator class is an example of poor maintainability as well, because every time a new command is added, all the interface must be adjusted to prevent text chaos in the terminal. I want to emphasize, that I created this class ONLY to make the experience of the game a little better by developing a pattern of how the text must be printed.

- Cohesion:
  - Although the TextCreator class is an example of a poorly maintainable class, it is a highly cohesive class, because each method of this class is responsible for printing only a particular text. These methods have no additional functionality.

How to play

1. Create a character by entering your preferred gender.
2. Go to the cabin to get the task and initial items.
3. Travel to the East and West islands to get stronger weapons and better food. Beat all the mobs.
4. Go to the North Island and defeat the boss.
5. Pick up the chest and go out.

Purpose of a command

go: Used to go to the next available place.
quit: Used to quit the game.
help: Used to get all the available commands.
hit: Used to hit a mob.
eat: Used to consume a food product and increment health.
look: Used to look around the current place and get information about items that are in that place.
take: Used to pick up an item that is in the player‘s current room.
drop: Used to drop an item from the backpack at the player's current location.
info: Used to get the name of the current place and all available exits.
list; Used to get a list of items that are in the backpack.
stats: Used to get health and backpack stats.
equip: Used to equip a weapon that is in the backpack.
about Used to get a description of an item that is in the backpack or in the current room. It can also be used to get a description of a mob that is currently in that room.
show: Used to get information about the weapon that is currently equiped.

How to use a command

go: > go [place] (exactly as it is written);
quit: > quit;
help: > help;
hit: > hit [mob's name];
eat: > eat [food item's name];
look: > look;
take: > take [item's name];
drop: > drop [item's name];
info: > info;
list; > list;
stats: > stats;
equip: > equip [item's name];
about: > about [item's name];
show: > show;

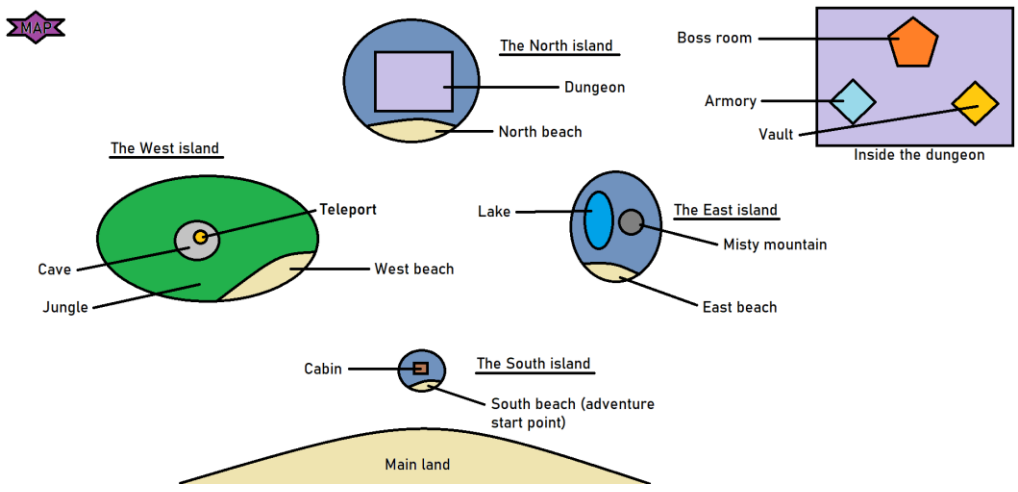
Bugs and problems

To my knowledge, there are no existing bugs in the game.

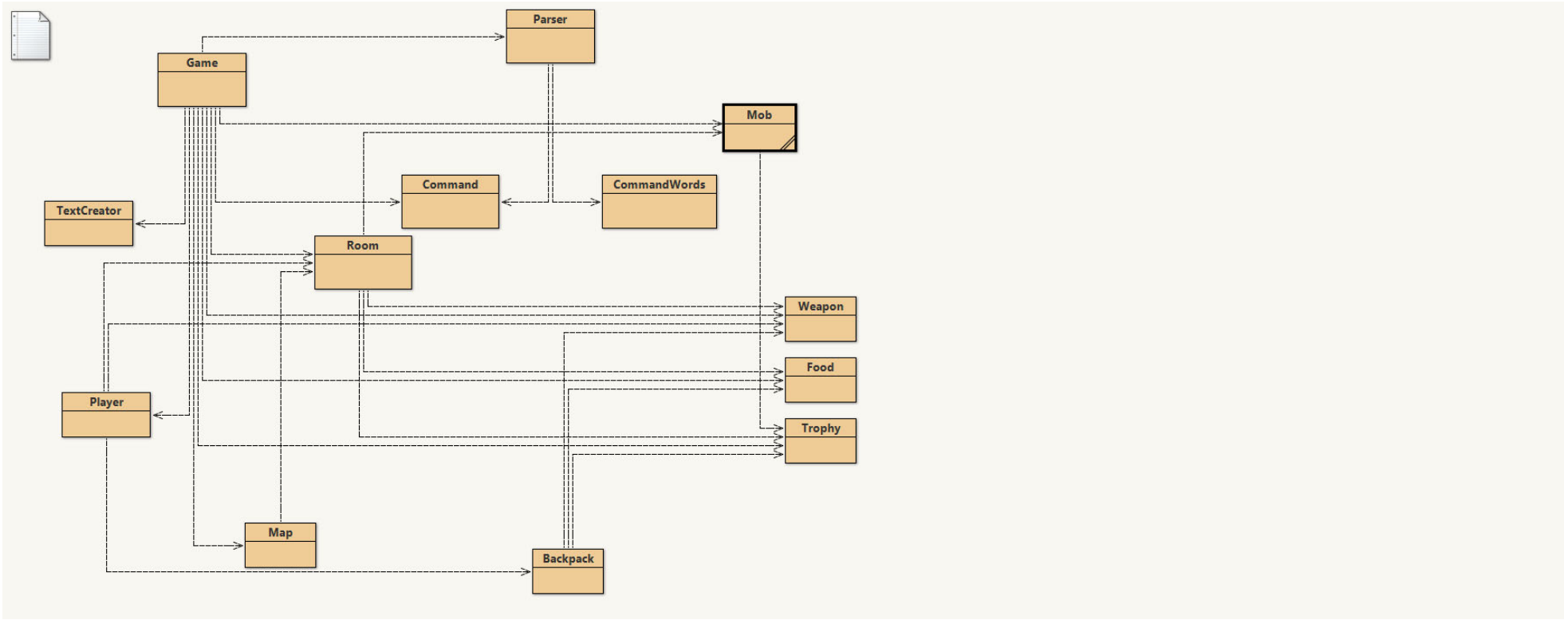
However, there are ample places where I could have written better code, but I didn‘t because of the lack of time and knowledge.

- TextCreator class methods have to be adjusted every time a new command is inserted or removed.
- If the text line is long enough, it goes out of bounds in the created text pattern.
- Some item names cannot be correctly printed out. For example a plural form of a word cherry, will be represented as cherrys, because by default if there is more than one item of the same kind, an s is added to the end of the word.

Map



For the initial code functionality, the credit goes to Michael Kölling and David J. Barnes.


Parser
Game
Mob
Command
CommandWords
TextCreator
Room
Weapon
Food
Player
Trophy
Map
Backpack

```
import java.util.Random;
/**
 * Class Mob - a mob in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Mob" represents one single mob in the game. Mobs can be found in different
places.
 * A player has to fight mobs to get items, trophies and finally, win the game.
 *
 * @author Vakaris Paulavičius (Student number: 20062023)
 * @version 2020.11.26
 */
public class Mob
{
    private String name;
    private Trophy trophy;
    private int health;
    private int maxHealth;
    private int minDamage;
    private int maxDamage;
    private boolean alive;
    private int chanceToDropTrophy;
    private String description;

    /**
     * Create new mob.
     *
     * @param name Mob's name.
     * @param trophy A Trophy object, that this mob can drop.
     * @param health Mob's initial health.
     * @param minDamage Minimum amount of damage this mob does.
     * @param minDamage Maximum amount of damage this mob does.
     * @param chanceToDropTrophy The chance (0-100%) that this mob will drop the
trophy when killed.
     */
    public Mob(String name, Trophy trophy, int maxHealth, int minDamage, int
maxDamage, int chanceToDropTrophy)
    {
        this.name = name;
        this.trophy = trophy;
        health = maxHealth;
        this.maxHealth = maxHealth;
        this.minDamage = minDamage;
        this.maxDamage = maxDamage;
        this.chanceToDropTrophy = chanceToDropTrophy;
        alive = true;
        setDescription();
    }

    /**
     * This method is used to get the name of the mob.
     * @return mob name.
```

```
     */
    public String getName()
    {
        return name;
    }

    /**
     * This method is used to get the current health of the mob.
     * @return current mob health.
     */
    public int getHealth()
    {
        return health;
    }

    /**
     * This method is used to set the current health to a particular amount.
     * If the mob is already dead, set its status to alive and set the current
health.
     *
     * @param amount Amount which the health is set to.
     */
    public void setHealth(int amount)
    {
        this.health = amount;
        if(!alive)
        {
            alive = true;
        }
         setDescription();
    }

    /**
     * This method is used to get the maximum health of the mob.
     * @return maximum mob health.
     */
    public int getMaxHealth()
    {
        return maxHealth;
    }

    /**
     * This method is used to get the trophy this mob drops.
     * @return trophy this mob drops
     */
    public Trophy getTrophy()
    {
        return trophy;
    }

    /**
     * This methodis used to set the mob dead
     */
    public void setDead()
```

```
{
    alive = false;
}

/**
 * This method returns the state of the mob.
 * @return true if alive, false if dead.
 */
public boolean isAlive()
{
    return alive;
}

/**
 * This method is used to reduce mob's current health by the amount in parameter.
 * @param amount An amount by which to decrement mob's health.
 */
public void reduceHealth(int amount)
{
    if((health-amount) > 0)
    {
        health -= amount;
    }
    else
    {
        setDead();
    }
    setDescription();
}

/**
 * This method is used to get a random damage this mob does from the range:
(minDamage-maxDamage).
 * @return random amount of damage in a single hit.
 */
public int getDamage()
{
    Random damageGenerator = new Random();
    int damage = damageGenerator.nextInt(maxDamage-minDamage) + minDamage;
    return damage;
}

/**
 * This method is used to get the minimum damage this mob does.
 * @return mob minimum damage.
 */
public int getMinDamage()
{
    return minDamage;
}

/**
 * This method is used to get the maximum damage this mob does.
 * @return mob maximum damage.
```

```
     */
    public int getMaxDamage()
    {
        return maxDamage;
    }

    /**
     * This method calculates whether this mob drops trophy when killed.
     * @return true if drops the trophy, false if does not.
     */
    public boolean dropsTrophy()
    {
        Random prob = new Random();
        int number = prob.nextInt(100);

        if((number + 1) <= chanceToDropTrophy)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    /**
     * This method is used to get the description of this mob.
     * @return mob description
     */
    public String getDescription()
    {
        return description;
    }

    // ***** PRIVATE METHODS *****

    /**
     * This method is used to set the description for the mob.
     */
    private void setDescription()
    {
        String line1 = ("Name: " + name),
               line2 = ("Damage: " + minDamage + "-" + maxDamage),
               line3 = ("Trophy that this mob might drop: " + trophy.getName()),
               line4 = ("Current health: " + health);

        description = line1 + "\n " + line2 + "\n " + line3 + "\n " + line4;
    }
}
```

```java
/**
 * Class Trophy - a trophy in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Trophy" represents one single trophy item in the game that a particular mob can
drop.
 * All mobs can drop different trophies which, most of the time, are just souvenirs.
 *
 * However, certain trophies are relevant to the game story and a player
 * must have them to win the game.
 *
 * @author Vakaris Paulavičius (Student number: 20062023)
 * @version 2020.11.26
 */
public class Trophy
{
    private String description;
    private String name;
    private int size;
    private boolean mainTrophy;

    /**
     * Create a new trophy.
     *
     * @param name Name of the trophy.
     * @param size How much space units this trophy occupies.
     */
    public Trophy(String name, int size)
    {
        this.name = name;
        this.size = size;
        setDescription();
    }

    /**
     * This method is used to get the name of the trophy.
     * @return trophy name.
     */
    public String getName()
    {
        return name;
    }

    /**
     * This method is used to get the size of this trophy.
     * @return how much space units it covers.
     */
    public int getSize()
    {
        return size;
    }
```

```
    /**
     * This method is used to get the description of this trophy.
     * @return trophy description
     */
    public String getDescription()
    {
        return description;
    }

    // ***** PRIVATE METHODS *****

    /**
     * This method is used to set a new description for this trophy.
     */
    private void setDescription()
    {
        String line1 = ("Name: " + name),
               line2 = ("Size in backpack: " + size);

        description = line1 + "\n " + line2;
    }
}
```

```java
import java.util.Random;
/**
 * Class Player - a player in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Player" object represents a character that a user is playing as.
 * It has basic attributes such as name, health and status.
 * An object of this class also follows it's current weapon, current and
 * previous locations. Every player instance has a backpack of a particular
 * size which is determined when creating a player.
 *
 * @author Vakaris Paulavičius (Student number: 20062023)
 * @version 2020.11.26
 */
public class Player
{
    // instance variables - replace the example below with your own
    private String name;
    private final int maxHealth = 100;
    private int currentHealth;
    boolean alive;

    private Room currentRoom;
    private Room previousRoom;
    private Weapon currentWeapon;

    private Backpack backpack;

    /**
     * Create a new player.
     *
     * @param gender Player's gender according to which a name is created.
     */
    public Player(String gender, Room currentRoom, int backpackSize)
    {
        name = setName(gender);  //Giving a name according to the provided gender
        currentHealth = 100;
        alive = true;
        this.currentRoom = currentRoom;
        previousRoom = null;
        currentWeapon = null;
        backpack = new Backpack(backpackSize);
    }

    /**
     * This method is used to get the name of the player object.
     * @return player's name.
     */
    public String getName()
    {
        return name;
    }
```

```java
/**
 * This method is used to get player's current health
 * @return player's current health.
 */
public int getHealth()
{
    return currentHealth;
}
```

```java
/**
 * This method is used to get player's backpack.
 * @return the backpack of this player.
 */
public Backpack getBackpack()
{
    return backpack;
}
```

```java
/**
 * This method is used to get player's maximum health
 * @return player maximum health.
 */
public int getMaxHealth()
{
    return maxHealth;
}
```

```java
/**
 * This method checks if the player is alive.
 * @return true if alive, false if dead
 */
public boolean isAlive()
{
    return alive;
}
```

```java
/**
 * This method sets the player's status dead.
 */
public void setDead()
{
    alive = false;
}
```

```java
/**
 * This method is used to increment player's health.
 *
 * @param amount Amount by which to increment player's health.
 */
public void incrementHealth(int amount)
{
    if((currentHealth + amount) <= 100)
    {
```

```
            currentHealth += amount;
        }
        else
        {
            currentHealth = maxHealth;
        }
    }

    /**
     * This method is used to decrement player's health.
     *
     * @param amount Amount by which to decrement player's health.
     */
    public void reduceHealth(int amount)
    {
        if((currentHealth - amount) > 0)
        {
            currentHealth -= amount;
        }
        else
        {
            setDead();
        }
    }

    /**
     * This method is used to get player's health details.
     * @return player's current health / player's maximum health.
     */
    public String getHealthStats()
    {
        return currentHealth + "/" + maxHealth;
    }

    /**
     * This method is used to change the room a player currently is in.
     *
     * @param newRoom A new Room the player is entering.
     */
    public void setCurrentRoom(Room newRoom)
    {
        currentRoom = newRoom;
    }

    /**
     * This method is used to change player's previous room.
     *
     * @param newRoom A Room the player has left.
     */
    public void setPreviousRoom(Room newRoom)
    {
        previousRoom = newRoom;
    }
```

```java
/**
 * This method is used to set the player's current weapon to the new one.
 *
 * @param newWeapon A Weapon the player has just equiped.
 */
public void setCurrentWeapon(Weapon newWeapon)
{
    currentWeapon = newWeapon;
}

/**
 * This method is used to get player's current weapon.
 * @return player's current weapon.
 */
public Weapon getCurrentWeapon()
{
    return currentWeapon;
}

/**
 * This method is used to get player's current room.
 * @return the room player currently is in.
 */
public Room getCurrentRoom()
{
    return currentRoom;
}

/**
 * This method is used to get player's previous room.
 * @return the room player was in before the current room.
 */
public Room getPreviousRoom()
{
    return previousRoom;
}
//******* PRIVATE METHODS *******

/**
 * This method is used to pick a random name according to the gender.
 * @return randomly generated name.
 */
private String setName(String gender)
{
    if(gender.toLowerCase().equals("male")) return randomMaleName();
    else if(gender.toLowerCase().equals("female")) return randomFemaleName();
    return null;
}

/**
 * This method is used to pick a random male name from the array.
 * @return randomly chosen male name.
 */
private String randomMaleName()
```

```
    {
        Random rand = new Random();
        String names[] = {"John Zena", "Captain Africa", "Bill Doors", "Elon Mist",
"Jackie Jack"};
        return names[rand.nextInt(names.length)];
    }

    /**
     * This method is used to pick a random female name from the array.
     * @return randomly chosen female name.
     */
    private String randomFemaleName()
    {
        Random rand = new Random();
        String names[] = {"Marilyn Momrow", "Super lady", "Marie Curry", "Hillmary
Cliffton", "Angelina Jokelee"};
        return names[rand.nextInt(names.length)];
    }
}
```

```java
/**
 * This class is part of the "Peacemaker" application.
 * "Peacemaker" is a 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 Vakaris Paulavičius (Student number:
20062023).
 * @version 2020.11.26
 */
public class CommandWords
{
    // a constant array that holds all valid command words
    private static final String[] validCommands = {
        "go", "quit", "help", "hit", "eat", "look", "take", "drop",
        "info", "list", "stats", "equip", "about", "show"
    };

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

    /**
     * Check whether a given String is a valid command word.
     * @param aString first word from the terminal's input.
     * @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;
    }

    /**
     * This method is used to get all valid commands.
     * @return all valid commands in a string.
     */
    public String getAll()
    {
        String str = "";
        for(String command: validCommands) {
            str += (command + "  ");
        }
        return str;
    }
}
```

```
}
```

```java
/**
 * Class Weapon - a weapon in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Weapon" represents one single weapon in the game that can be used
 * in fights. Game characters can acquire, use and possess weapons. Weapons can
 * also be found in rooms.
 *
 * @author Vakaris Paulavičius (Student number: 20062023)
 * @version 2020.11.26
 */
public class Weapon
{
    private String name;
    private int damage;
    private String rarity;
    private String type;
    private int size;
    private String description;

    private String[] rarityTypes = {"Common", "Rare", "Almost infeasible to find"};
//Should be changed to a class in the future
    private String[] weaponTypes = {"melee", "ranged"};  //Should be changed to a
class in the future

    /**
     * Create a new weapon.
     *
     * @param name Weapon's name.
     * @param damage How much damage this weapon does in one hit.
     * @param rarity Weapon's rarity (0 - common, 1 - rare, 2 - ultra rare).
     * @param size Size of the weapon (how much space it covers).
     * @param type Weapon's type (0 - melee, 1 - ranged).
     */
    public Weapon(String name, int damage, int rarity, int size, int type)
    {
        this.name = name;
        this.damage = damage;
        this.rarity = rarityTypes[rarity];
        this.type = weaponTypes[type];
        this.size = size;
        setDescription();
    }

    /**
     * This method returns the size of the weapon.
     * @return how much space the weapon occupies.
     */
    public int getSize()
    {
        return size;
    }
```

```java
/**
 * This method returns weapon's rarity.
 * @return weapon's rarity.
 */
public String getRarity()
{
    return rarity;
}

/**
 * This method is used to get the damage this weapon does.
 * @return weapon damage.
 */
public int getDamage()
{
    return damage;
}

/**
 * This method is used to get the type of the weapon.
 * @return weapon type.
 */
public String getType()
{
    return type;
}

/**
 * This method is used to get the name of the weapon.
 * @return weapon name.
 */
public String getName()
{
    return name;
}

/**
 * This method is used to get the description of the weapon.
 * @return weapon description.
 */
public String getDescription()
{
    return description;
}

// ***** PRIVATE METHODS *****

/**
 * This method is used to set a new description for this object.
 */
private void setDescription()
{
    String line1 = ("Name: " + name),
```

```
            line2 = ("Damage: " + damage),
            line3 = ("Size in backpack: " + size),
            line4 = ("Rarity: " + rarity),
            line5 = ("Type: " + type);

        description = line1 + "\n " + line2 + "\n " + line3 + "\n " + line4 + "\n " +
line5;
    }
}
```

```
import java.util.Scanner;
/**
 * Class Parser - user's input analyser in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Parser" has a purpose of reading the player's input from the terminal,
 * and determing what are the further actions.
 *
 * @author Michael Kölling, David J. Barnes and Vakaris Paulavičius (Student number:
20062023).
 * @version 2020.11.26
 */
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);
    }

    /**
     * This method is used to get the next command from the user.
     * @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
                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);
    }
}

/**
 * This method is used to get all valid commands.
 * @return a string with all command that are valid.
 */
public String getCommands()
{
    return commands.getAll();
}
}
```

```java
import java.util.HashMap;
/**
 * Class Backpack - a player's backpack in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Backpack" represents one single backpack where player's trophies,
 * weapons and food are stored. It has limited space. and is made out of
 * three HashMaps: for food, for weapons and for trophies.
 *
 * @author Vakaris Paulavičius (Student number: 20062023).
 * @version 2020.11.26
 */
public class Backpack
{
    private HashMap<Food, Integer> foodList;        //stores food that is in this backpack
    private HashMap<Weapon, Integer> weapons;           //stores weapons that are in this backpack
    private HashMap<Trophy, Integer> trophies;          //stores trophies that are in this backpack

    private int maxBackpackSize;
    private int currentBackpackSize;

    /**
     * Create a backpack.
     *
     * @param maxSize Maximum size of the backpack
     */
    public Backpack(int maxSize)
    {
        foodList = new HashMap<>();
        weapons = new HashMap<>();
        trophies = new HashMap<>();

        maxBackpackSize = maxSize;
        currentBackpackSize = 0;
    }

    /**
     * This method is used to get the current backpack size.
     * @return backpack's current size
     */
    public int getCurrentBackpackSize()
    {
        return currentBackpackSize;
    }

    /**
     * This method is used to get the maximum backpack size.
     * @return backpack's maximum size
     */
```

```
public int getMaxBackpackSize()
{
    return maxBackpackSize;
}

/**
 * This method is used to add a weapon to the backpack.
 *
 * @param item Weapon to be added.
 * @return true, if the weapon can be added, false if it cannot be added.
 */
public boolean addWeapon(Weapon item)
{
    if(inWeapons(item))
    {
        return false;  //A player can't have more than one weapon of the same
kind
    }
    else
    {
        if(isSpaceInBackpack(item.getSize()))
        {
            weapons.put(item, 1);
            currentBackpackSize += item.getSize();
            return true;
        }
        else
        {
            return false;
        }
    }
}

/**
 * This method is used to add a food object to the backpack.
 *
 * @param item Food product to be added.
 * @return true, if the food product can be added, false if it cannot be added.
 */
public boolean addFood(Food item)
{
    if(inFoodList(item))
    {
        if(isSpaceInBackpack(item.getSize()))
        {
            int quantity = foodList.get(item);
            foodList.put(item, quantity +1);
            currentBackpackSize += item.getSize();
            return true;
        }
        else
        {
            return false;
        }
```

```
        }
        else
        {
            if(isSpaceInBackpack(item.getSize()))
            {
                foodList.put(item, 1);
                currentBackpackSize += item.getSize();
                return true;
            }
            else
            {
                return false;
            }
        }
    }

    /**
     * This method is used to add a trophy to the backpack.
     *
     * @param item Trophy to be added.
     * @return true, if the trophy can be added, false if cannot it be added.
     */
    public boolean addTrophy(Trophy item)
    {
        if(inTrophies(item))
        {
            if(isSpaceInBackpack(item.getSize()))
            {
                int quantity = foodList.get(item);
                trophies.put(item, quantity +1);
                currentBackpackSize += item.getSize();
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            if(isSpaceInBackpack(item.getSize()))
            {
                trophies.put(item, 1);
                currentBackpackSize += item.getSize();
                return true;
            }
            else
            {
                return false;
            }
        }
    }

    /**
```

```java
 * This method is used to remove a trophy from the backpack.
 *
 * @param item Trophy to be removed.
 * @return true, if the trophy is in the backpack, false if it is not.
 */
public boolean removeTrophy(Trophy item)
{
    if(inTrophies(item))
    {
        if(trophies.get(item) == 1)
        {
            trophies.remove(item);
            currentBackpackSize -= item.getSize();
        }
        else
        {
            int quantity = trophies.get(item);
            trophies.put(item, quantity -1);
            currentBackpackSize -= item.getSize();
        }
        return true;
    }
    else
    {
        return false;
    }
}

/**
 * This method is used to remove a food object from the backpack.
 *
 * @param item Food object to be removed.
 * @return true, if the food object is in the backpack, false if it is not.
 */
public boolean removeFood(Food item)
{
    if(inFoodList(item))
    {
        if(foodList.get(item) == 1)
        {
            foodList.remove(item);
            currentBackpackSize -= item.getSize();
        }
        else
        {
            int quantity = foodList.get(item);
            foodList.put(item, quantity -1);
            currentBackpackSize -= item.getSize();
        }
        return true;
    }
    else
    {
        return false;
```

```
        }
    }

    /**
     * This method is used to remove a weapon from the backpack.
     *
     * @param item Weapon to be removed.
     * @return true, if the weapon is in the backpack, false if it is not.
     */
    public boolean removeWeapon(Weapon item)
    {
        if(inWeapons(item))
        {
            if(weapons.get(item) == 1)
            {
                weapons.remove(item);
                currentBackpackSize -= item.getSize();
            }
            else
            {
                int quantity = weapons.get(item);
                weapons.put(item, quantity -1);
                currentBackpackSize -= item.getSize();
            }
            return true;
        }
        else
        {
            return false;
        }
    }

    /**
     * This method is used to get a food object from the backpack using its name.
     *
     * @param name Food object's name.
     * @return food object if it is in the backpack, null if the object was not
found.
     */
    public Food getFoodByName(String name)
    {
        for(Food key : foodList.keySet())
        {
            if(key.getName().toLowerCase().equals(name)) return key;
        }
        return null;
    }

    /**
     * This method is used to get a weapon object from the backpack using its name.
     *
     * @param name Weapon's name.
     * @return weapon object if it is in the backpack, null if the object was not
found.
```

```
     */
    public Weapon getWeaponByName(String name)
    {
        for(Weapon key : weapons.keySet())
        {
            if(key.getName().toLowerCase().equals(name)) return key;
        }
        return null;
    }

    /**
     * This method is used to get a trophy object from the backpack using its name.
     *
     * @param name Trophy's name
     * @return trophy object if it is in the backpack, null if the object was not
found.
     */
    public Trophy getTrophyByName(String name)
    {
        for(Trophy key : trophies.keySet())
        {
            if(key.getName().toLowerCase().equals(name)) return key;
        }
        return null;
    }

    /**
     * This method is used to get all the trophies that are in the backpack.
     * @return a string with all the trophies that are in the backpack.
     */
    public String listTrophies()
    {
        if(trophies.isEmpty())
        {
            return "None.";
        }
        else
        {
            int index = 0;
            String str = "";
            for(Trophy key: trophies.keySet())
            {
                if(index == (trophies.size()-1))
                {
                    if(trophies.get(key) == 1)
                    {
                        str += (trophies.get(key) + " " + key.getName() + ".");
                    }
                    else
                    {
                        str += (trophies.get(key) + " " + key.getName() + "s.");
                    }
                }
                else
```

```
            {
                if(trophies.get(key) == 1)
                {
                    str += (trophies.get(key) + " " + key.getName() + ", ");
                }
                else
                {
                    str += (trophies.get(key) + " " + key.getName() + "s, ");
                }
            }
            index ++;
        }
        return str;
    }
}

/**
 * This method is used to get all the food products that are in the backpack.
 * @return a string with all the food products that are in the backpack.
 */
public String listFood()
{
    if(foodList.isEmpty())
    {
        return "None.";
    }
    else
    {
        int index = 0;
        String str = "";
        for(Food key: foodList.keySet())
        {
            if(index == (foodList.size()-1))
            {
                if(foodList.get(key) == 1)
                {
                    str += (foodList.get(key) + " " + key.getName() + ".");
                }
                else
                {
                    str += (foodList.get(key) + " " + key.getName() + "s.");
                }
            }
            else
            {
                if(foodList.get(key) == 1)
                {
                    str += (foodList.get(key) + " " + key.getName() + ", ");
                }
                else
                {
                    str += (foodList.get(key) + " " + key.getName() + "s, ");
                }
            }
```

```
            index ++;
        }
        return str;
    }
}

/**
 * This method is used to get all the weapons that are in the backpack.
 * @return a string with all the weapons that are in the backpack.
 */
public String listWeapons()
{
    if(weapons.isEmpty())
    {
        return "None.";
    }
    else
    {
        int index = 0;
        String str = "";
        for(Weapon key: weapons.keySet())
        {
            if(index == (weapons.size()-1))
            {
                if(weapons.get(key) == 1)
                {
                    str += (key.getName() + ".");
                }
                else
                {
                    str += (key.getName() + "s.");
                }
            }
            else
            {
                if(weapons.get(key) == 1)
                {
                    str += (key.getName() + ", ");
                }
                else
                {
                    str += (key.getName() + "s, ");
                }
            }
            index ++;
        }
        return str;
    }
}

/**
 * This method is used to check if the main trophy is in the backpack.
 * Used to determine whether the player has won.
```

```
     * @return true if the main trophy is in the player's backpack, false if it is
not.
     */
    public boolean hasMainTrophy(Trophy trophy)
    {
        if(trophies.containsKey(trophy)) return true;
        else return false;
    }

    /**
     * This method is used to get backpack details.
     * @return current backpack size / maximum backpack size
     */
    public String getBackpackStats()
    {
        return getCurrentBackpackSize() + "/" + getMaxBackpackSize();
    }

    //******* PRIVATE METHODS *******

    /**
     * Checks whether the weapon is already in the backpack.
     *
     * @param item Weapon object.
     * @return true if it is, false otherwise.
     */
    private boolean inWeapons(Weapon item)
    {
        return weapons.containsKey(item);
    }

    /**
     * Checks whether the trophy is already in the backpack.
     *
     * @param item Trophy object.
     * @return true if it is, false otherwise.
     */
    private boolean inTrophies(Trophy item)
    {
        return trophies.containsKey(item);
    }

    /**
     * Checks whether the food object is already in the backpack.
     *
     * @param item Food object.
     * @return true if it is, false otherwise.
     */
    private boolean inFoodList(Food item)
    {
        return foodList.containsKey(item);
    }

    /**
```

```
    * Checks if there is enough space in the backpack for a particular item.
    * @param size Item's size.
    */
    private boolean isSpaceInBackpack(int size)
    {
        if((currentBackpackSize + size) <= maxBackpackSize)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.Random;
/**
 * Class Game is the main class of this application.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A Game object is used to simulate the actual game. It creates a map,
 * a character and all other objects. It also considers the input of a player
 * and executes all the instructions.
 *
 * @author Michael Kölling, David J. Barnes and Vakaris Paulavičius (Student number:
20062023).
 * @version 2020.11.28
 */

public class Game
{
    private Parser parser;
    private Trophy goal;
    private Map map;
    private Player player;
    private TextCreator text;

    /**
     * Create a game.
     */
    public Game()
    {
        createMap();
        parser = new Parser();
        text = new TextCreator();
    }

    /**
     * This method is used to start the game.
     * @param args An empty string array.
     */
    public static void main(String args[])
    {
        Game game = new Game();
        game.play();
    }

    // ***** PRIVATE METHODS *****

    /**
     *  Main play routine. Creates a player and then loops until the player either
wins the game or dies or enters
     *  the command 'quit'.
     */
    private void play()
```

```
{
    text.printWelcome();
    boolean finished = false;

    //Character creation
    if(!createCharacter())
    {
        finished = true;
    }
    else
    {
        text.printTopBox();
        text.classicPattern(player.getCurrentRoom().getLongDescription());
    }

    //Main game loop
    while (! finished) {
        Command command = parser.getCommand();
        finished = processCommand(command);

        if(player.getCurrentRoom() == map.getTaskRoom() &&
!map.taskRoomWasVisited())
        {
            text.taskMessage();
            map.visitTaskRoom();
        }

        if(!player.isAlive())
        {
            text.deathText();
            finished = true;
        }

        if(playerWon())
        {
            finished = true;
        }
    }

    text.printBottomBox();
    text.printGoodbye();
}

/**
 * This method processes the given command.
 *
 * @param command The command to be processed.
 * @return true If the command ends the game, false otherwise.
 */
private boolean processCommand(Command command)
{
    boolean wantToQuit = false;

    if(command.isUnknown()) {
```

```
            text.classicPattern("There is no such command, sir.");
            return false;
        }

        String commandWord = command.getCommandWord();  //I am using switch to
prevent a ton of else if's
        switch(commandWord)
        {
            case "help":
            text.printHelp(parser.getCommands());
            break;

            case "go":
            goRoom(command);
            break;

            case "quit":
            wantToQuit = quit(command);
            break;

            case "look":
            text.printCurrentRoomDetails(player.getCurrentRoom().listMobs(),
player.getCurrentRoom().listFood(),
                player.getCurrentRoom().listWeapons(),
player.getCurrentRoom().listTrophies());
            break;

            case "info":
            text.classicPattern(player.getCurrentRoom().getLongDescription());
            break;

            case "eat":
            eatFood(command);
            break;

            case "equip":
            equipWeapon(command);
            break;

            case "list":
            text.backpackItems(player.getBackpack().listFood(),
player.getBackpack().listWeapons(), player.getBackpack().listTrophies());
            break;

            case "stats":
            text.line(" Backpack: " + player.getBackpack().getBackpackStats());
            text.classicPattern("Health: " + player.getHealthStats());;
            break;

            case "drop":
            dropItem(command);
            break;

            case "take":
```

```
            takeItem(command);
            break;

            case "hit":
            hitMob(command);
            break;

            case "show":
            weaponInfo();
            break;

            case "about":
            aboutObject(command);
            break;
        }
        // else command not recognised.
        return wantToQuit;
    }

    /**
     * This method is used whenever user's command is 'go'. It checks if the player
can go to his specified direction
     * and performs actions according to the command. Also, prints appropriate text
to the terminal.
     *
     * @param command The command user has input.
     */
    private void goRoom(Command command)
    {
        if(!command.hasSecondWord()) {
            // if there is no second word, we don't know where to go...
            text.classicPattern("Go where?");
            return;
        }

        String direction = command.getSecondWord();

        // Try to leave current room.
        Room nextRoom = player.getCurrentRoom().getExit(direction);
        // Check for mobs in that room
        if(player.getCurrentRoom().hasMobs())
        {
            text.classicPattern("You cannot go to another room unless you have killed
all the mobs!");
            return;
        }

        if(player.getPreviousRoom() == null && direction.equals("back"))
        {
            text.classicPattern("This is the starting point, sir. You cannot go
back!");
        }
        else if(player.getPreviousRoom() == map.getStartRoom() &&
direction.equals("back"))
```

```
        {
            text.classicPattern("You cannot go back once you've started the
adventure.");
        }
        else if(player.getPreviousRoom() == player.getCurrentRoom() &&
direction.equals("back"))
        {
            text.classicPattern("You've just went back.");
        }
        else if(player.getPreviousRoom() != null && direction.equals("back"))
        {
            player.setCurrentRoom(player.getPreviousRoom());
            text.classicPattern(player.getPreviousRoom().getLongDescription());
        }
        else if(nextRoom == null)
        {
            text.classicPattern("There is no such place!");
        }
        else if(nextRoom == map.getTeleportRoom())
        {
            teleportToRoom();
        }
        else {
            player.setPreviousRoom(player.getCurrentRoom());
            player.setCurrentRoom(nextRoom);
            text.classicPattern(player.getCurrentRoom().getLongDescription());
        }
    }

    /**
     * This method is used to get information about the player's current weapon.
     */
    private void weaponInfo()
    {
        if(player.getCurrentWeapon() != null)
        {
            text.classicPattern("Your current weapon is " +
player.getCurrentWeapon().getName() + ". It does " +
player.getCurrentWeapon().getDamage() + " damage.");
        }
        else
        {
            text.classicPattern("You currently have no weapon in your hand.");
        }
    }

    /**
     * This method is used to choose a random room which teleport the player to
     */
    private void teleportToRoom()
    {
        player.setPreviousRoom(player.getCurrentRoom());
        player.setCurrentRoom(map.teleportToRoom());
        text.classicPattern(player.getCurrentRoom().getLongDescription());
```

```
    }

    /**
     * This method is used to consume food item. It checks if the user's specified
food is in his backpack.
     * If yes, it is removed from it and player's health is increased.
     *
     * @param command The command user has input.
     */
    private void eatFood(Command command)
    {
        String name = getObjectName(command);  //Gets object's name according to the
entered text line

        Food food = player.getBackpack().getFoodByName(name);
        if(food != null)
        {
            player.getBackpack().removeFood(food);
            player.incrementHealth(food.getValue());
            if(player.getHealth() == player.getMaxHealth())
            {
                text.classicPattern("You ate " + food.getName() + ". Now your health
is full.");
            }
            else
            {
                text.classicPattern("You ate " + food.getName() + " and incremented
your health by " + food.getValue() + ".");
            }
        }
        else
        {
            text.classicPattern("You have either misspelled the food name or there is
no such food in your backpack.");
        }
    }

    /**
     * This method is used to equip a weapon. It checks if the user's specified
weapon is in his backpack.
     * If yes, new weapon is equiped, if not, an error message is printed out to the
terminal.
     *
     * @param command The command user has input.
     */
    private void equipWeapon(Command command)
    {
        String name = getObjectName(command);
        Weapon weapon = player.getBackpack().getWeaponByName(name);
        if(weapon != null)
        {
            player.setCurrentWeapon(weapon);
            text.classicPattern("You have taken the " + weapon.getName() + ".");
        }
```

```
        else
        {
            text.classicPattern("You either made a typing mistake or there is no such
weapon in your backpack.");
        }
    }

    /**
     * This method is used to get information about an item. It checks whether the
item is in the user's
     * backpack or in the current room. If it is found in one of those, it's
description is printed out to the terminal.
     * Else, an error message is printed out (player can only get description of
items that are either in his backpack or
     * in the current room).
     *
     * @param command The command user has input.
     */
    private void aboutObject(Command command)
    {
        String name = getObjectName(command);

        //Check whether the item is food and is it in this room
        Food rFood = player.getCurrentRoom().getFoodByName(name);
        if(rFood != null)
        {
            text.classicPattern(rFood.getDescription());
            return;
        }
        //Check whether the item is food and is it in the backpack
        Food bFood = player.getBackpack().getFoodByName(name);
        if(bFood != null)
        {
            text.classicPattern(bFood.getDescription());
            return;
        }

        //Check whether the item is a weapon and is it in this room
        Weapon rWeapon = player.getCurrentRoom().getWeaponByName(name);
        if(rWeapon != null)
        {
            text.classicPattern(rWeapon.getDescription());
            return;
        }
        //Check whether the item is a weapon and is it in the backpack
        Weapon bWeapon = player.getBackpack().getWeaponByName(name);
        if(bWeapon != null)
        {
            text.classicPattern(bWeapon.getDescription());
            return;
        }

        //Check whether the item is a trophy and is it in this room
        Trophy rTrophy = player.getCurrentRoom().getTrophyByName(name);
```

```
        if(rTrophy != null)
        {
            text.classicPattern(rTrophy.getDescription());
            return;
        }
        //Check whether the item is a trophy and is it in the backpack
        Trophy bTrophy = player.getBackpack().getTrophyByName(name);
        if(bTrophy != null)
        {
            text.classicPattern(bTrophy.getDescription());
            return;
        }

        //Checks whether the object is mob and is it in this room
        Mob mob = player.getCurrentRoom().getMobByName(name);
        if(mob != null)
        {
            text.classicPattern(mob.getDescription());
            return;
        }

        //Prints this string if the specified object was not found in this room and
in the backpack
        if(bTrophy == null && rTrophy == null && bWeapon == null && rTrophy == null
&& rFood == null && bFood == null && mob == null)
        {
            text.classicPattern("There is no such object neither in this room nor in
your backpack.");
        }
    }

    /**
     * This method is used to drop an item from the backpack int he current room.
Player's input is checked to determine
     * whether the mentioned item is in backpack and what kind of item it is.
     * If it turns out, that the item is in the backpack, then it is removed from the
player's backpack and left in the current room,
     * else, an error message is printed out to the terminal.
     *
     * @param command The command user has input.
     */
    private void dropItem(Command command)
    {
        String name = getObjectName(command);
        //Check whether the item is food and is it in the backpack
        Food food = player.getBackpack().getFoodByName(name);
        if(food != null)
        {
            text.classicPattern("You have dropped " + food.getName() + " in this
room.");

            player.getBackpack().removeFood(food);
            player.getCurrentRoom().addFood(food, 1);
            return;
        }
```

```
        //Check whether the item is a weapon and is it in the backpack
        Weapon weapon = player.getBackpack().getWeaponByName(name);
        if(weapon != null)
        {
            text.classicPattern("You have dropped " + weapon.getName() + " in this
room.");
            player.getBackpack().removeWeapon(weapon);
            player.getCurrentRoom().addWeapons(weapon, 1);
            return;
        }
        //Check whether the item is a trophy and is it in the backpack
        Trophy trophy = player.getBackpack().getTrophyByName(name);
        if(trophy != null)
        {
            text.classicPattern("You have dropped " + trophy.getName() + " in this
room.");
            player.getBackpack().removeTrophy(trophy);
            player.getCurrentRoom().addTrophies(trophy);
            return;
        }
        //Prints this string if the specified item was not found in this room
        if(trophy == null && weapon == null && food == null)
        {
            text.classicPattern("You either made a typing mistake or there is no such
item in your backpack.");
        }
    }

    /**
     * This method is used to pick up an item from the current room. Player's input
is checked to determine
     * whether the mentioned item is in that room and what kind of item it is.
     * If it turns out, that the item is in the room, then it is added to player's
backpack (if there is enough space) and removed from
     * the current room, else, an error message is printed out to the terminal.
     *
     * @param command The command user has input.
     */
    private void takeItem(Command command)
    {
        //Checks whether the item's name consists of two words or one
        String name = getObjectName(command);
        //Check whether the item is food and is it in this room
        Food food = player.getCurrentRoom().getFoodByName(name);
        if(food != null)
        {
            if(player.getBackpack().addFood(food))
            {
                text.classicPattern("You have taken " + food.getName() + " from this
room.");
                player.getCurrentRoom().removeFood(food);
            }
            else
            {
```

```
            text.classicPattern("There is not enough space in your backpack.");
        }
        return;
    }
    //Check whether the item is food and is it in this room
    Weapon weapon = player.getCurrentRoom().getWeaponByName(name);
    if(weapon != null)
    {
        if(player.getBackpack().addWeapon(weapon))
        {
            text.classicPattern("You have taken " + weapon.getName() + " from
this room.");
            player.getCurrentRoom().removeWeapon(weapon);
        }
        else
        {
            text.classicPattern("You cannot have two weapons of the same kind.");
        }
        return;
    }
    //Check whether the item is food and is it in this room
    Trophy trophy = player.getCurrentRoom().getTrophyByName(name);
    if(trophy != null)
    {
        if(player.getBackpack().addTrophy(trophy))
        {
            text.classicPattern("You have taken " + trophy.getName() + " from
this room.");
            player.getCurrentRoom().removeTrophy(trophy);
        }
        else
        {
            text.classicPattern("There is not enough space in your backpack.");
        }
        return;
    }
    //Prints this string if the specified item was not found in this room
    if(trophy == null && weapon == null && food == null)
    {
        text.classicPattern("You either made a typing mistake or there is no such
item in this room.");
    }
}

/**
 * This method is used to to simulate a fight whenever user types a command 'hit
+ (mob name)'.
 * If there is a mob in the current room, that the user specified in his command,
appropriate actions are performed and
 * the right text is printed out to the terminal.
 *
 * @param command The command user has input.
 */
private void hitMob(Command command)
```

```
    {
        if(player.getCurrentRoom().hasMobs())
        {
            String name = getObjectName(command);
            Mob mob = player.getCurrentRoom().getMobByName(name);
            if(mob != null)
            {
                int damage = mob.getDamage();
                if(player.getCurrentWeapon() == null)
                {
                    noWeaponCase(mob, damage);
                    return;
                }
                else
                {
                    player.reduceHealth(damage);
                    mob.reduceHealth(player.getCurrentWeapon().getDamage());
                }

                if(!player.isAlive()) return;
                else
                {
                    text.fightText(player.getCurrentWeapon().getDamage(),
player.getHealthStats(), mob.getHealth(), damage, mob.getName(), mob.isAlive());
                }

                if(!mob.isAlive())
                {
                    player.getCurrentRoom().removeMob(mob);
                    if(mob.dropsTrophy())
                    {
                        player.getCurrentRoom().addTrophies(mob.getTrophy());
                        text.classicPattern("Huuuraaayyy!! " + mob.getName() + " has
dropped the " + mob.getTrophy().getName() + ".");
                    }
                }
            }
            else
            {
                text.classicPattern("There is no such mob in this place!");
            }
        }
        else
        {
            text.classicPattern("There are no mobs in this place... Are you seeing
ghosts already?");
        }
    }

    /**
     * This method is used to generate a fight text if a player does not have a
weapon in his hand.
     *
     * @param mob A mob the player is fighting.
```

```
     * @param damage An amount of damage this mob does.
     */
    private void noWeaponCase(Mob mob, int damage)
    {
        Random rand = new Random();
        String cheek;
        if(rand.nextInt(2) == 0)
        {
            cheek = "left cheek";
        }
        else
        {
            cheek = "right cheek";
        }

        text.line(" You slapped " + mob.getName() + "'s " + cheek + " and did 0
damage.");
        text.classicPattern(mob.getName() + " did " + damage + " damage to you!");
        player.reduceHealth(damage);
    }

    /**
     * This method is used to check whether the player has won.
     * @return true, if the player has won, false if not.
     */
    private boolean playerWon()
    {
        if(player.getCurrentRoom() == map.getEndRoom())
        {
            if(player.getBackpack().hasMainTrophy(goal))
text.winMessage(player.getName());
            else text.noMainTrophyMessage();
            return true;
        }
        else
        {
            return false;
        }
    }

    /**
     * "Quit" was entered. Check the rest of the command to see
     * whether we really quit the game.
     * @return true, if this command quits the game, false otherwise.
     */
    private boolean quit(Command command)
    {
        if(command.hasSecondWord()) {
            text.classicPattern("Quit what?");
            return false;
        }
        else {
            return true;  // signal that we want to quit
        }
```

```
}

/**
 * This method is used to create a character.
 * @return true if the creation was successfull, false otherwise.
 */
private boolean createCharacter()
{
    text.createCharacterText();
    Scanner reader = new Scanner(System.in);
    boolean selected = false;

    System.out.print("> ");
    String inputLine = reader.nextLine();
    if(genderExists(inputLine))
    {
        createPlayerObject(inputLine);
        return true;
    }
    else if(wordWasQuit(inputLine))
    {
        return false;
    }
    else
    {
        while(!selected)
        {
            text.classicPattern("Your command was not recognised. Please try
again.");  //print this if the player has failed to input a valid gender or a word
'quit'.

            System.out.print("> ");
            inputLine = reader.nextLine();
            if(genderExists(inputLine))
            {
                createPlayerObject(inputLine);
                return true;
            }
            else if(wordWasQuit(inputLine))
            {
                return false;
            }
        }
        return false;
    }
}

/**
 * This method creates a new player object.
 *
 * @param gender User's specified gender.
 */
private void createPlayerObject(String gender)
{
    player = new Player(gender, map.getStartRoom(), 40);
```

```
        text.characterText(player.getName());
    }

    /**
     * This method checks whether user has input a valid gender.
     *
     * @param gender User's specified gender.
     * @return true if the gender valid, false otherwise.
     */
    private boolean genderExists(String gender)
    {
        if(gender.trim().toLowerCase().equals("male") ||
gender.trim().toLowerCase().equals("female"))
        {
            return true;
        }
        return false;
    }

    /**
     * This method is used to check whether the word was 'quit'.
     *
     * @param word Word to check
     * @return true if the command was 'quit', false if it was not.
     */
    private boolean wordWasQuit(String word)
    {
        if(word.trim().toLowerCase().equals("quit"))
        {
            return true;
        }
        return false;
    }

    /**
     * This method checks whether the user input consists of two words or three and
returns appropriate string.
     *
     * @param command User's input fro the console.
     * @return item or mob name
     */
    private String getObjectName(Command command)
    {
        if(command.hasThirdWord())
        {
            return (command.getSecondWord() + " " +
command.getThirdWord()).toLowerCase();
        }
        else if(command.hasSecondWord())
        {
            return command.getSecondWord().toLowerCase();
        }
        else
        {
```

```
            return null;
        }
    }

    //Creating map and other objects

    /**
     * Create all the rooms, weapons, trophies, mobs and a backpack.
     */
    private void createMap()
    {
        Room wBeach, eBeach, nBeach, sBeach, cabin, cave, jungle, mountain, lake,
dungeon, armory, roof, chestRoom, wc, mainLand, teleport;

        // create the rooms
        wBeach = new Room("You are on the beach of the Entanglement Island.");
        eBeach = new Room("You are on the beach of the Elevation Island.");
        nBeach = new Room("You are on the beach of the Ruthlessness Island.");
        sBeach = new Room("You are on the beach of the Placidity Island.");
        cabin = new Room("You have arrived to the cabin of the old man.");
        cave = new Room("You've just sneeked into a dangerous cave.");
        jungle = new Room("You've just entered the puzzling jungle. Be careful and
don't get lost." + "\n " + "The place is full of cannibals and angry bees!");
        mountain = new Room("You have successfully reached the top of the Misty
Mountain." + "\n " + "Be aware of the mighty vultures!");
        lake = new Room("You arrived to the magic lake.");
        dungeon = new Room("Shhhhhh..... be quiet chief, you have just entered the
dungeon of the" + "\n " + "merciless Captain Toad.");
        armory = new Room("You arrived to the armory.");
        roof = new Room("You got on top of the dungeon.");
        chestRoom = new Room("You've just entered the captain room.");
        wc = new Room("Welcome to the captain's personal toilet. If your pants are
already brimming with fear," + "\n " + "rest here, or else, continue the journey like
a true warrior!");
        mainLand = new Room("You are on the main land.");
        teleport = new Room("");  //When a player gets to this room, he is teleported
to a random room from the array list : 'availableTeleportDestinations';

        Room allRooms[] = new Room[] {wBeach, eBeach, nBeach, sBeach, cabin, cave,
jungle, mountain, lake, dungeon, armory, wc, roof, chestRoom, mainLand};
        //Sets the available teleport destinations. Unfortunately, everytime a new
room is added, this array must be adjusted.
        Room roomsToTeleportTo[] = new Room[] {wBeach, eBeach, nBeach, sBeach, cabin,
cave, jungle, mountain, lake, dungeon, armory, wc};

        // initialise exits

        //South island
        sBeach.setExit("cabin", cabin);
        sBeach.setExit("west", wBeach);
        sBeach.setExit("east", eBeach);

        cabin.setExit("beach", sBeach);
```

```
        //East island
        eBeach.setExit("lake", lake);
        eBeach.setExit("mountain", mountain);
        eBeach.setExit("south", sBeach);
        eBeach.setExit("north", nBeach);

        lake.setExit("beach", eBeach);
        lake.setExit("mountain", mountain);

        mountain.setExit("lake", lake);
        mountain.setExit("beach", eBeach);

        //West island
        wBeach.setExit("south", sBeach);
        wBeach.setExit("north", nBeach);
        wBeach.setExit("jungle", jungle);

        cave.setExit("jungle", jungle);
        cave.setExit("teleport", teleport);

        jungle.setExit("cave", cave);
        jungle.setExit("beach", wBeach);

        //North island
        nBeach.setExit("east", eBeach);
        nBeach.setExit("west", wBeach);
        nBeach.setExit("dungeon", dungeon);

        dungeon.setExit("outside", nBeach);
        dungeon.setExit("vault", wc);
        dungeon.setExit("armory", armory);
        dungeon.setExit("boss", chestRoom);

        armory.setExit("back", dungeon);

        wc.setExit("back", dungeon);

        chestRoom.setExit("out", roof);

        mainLand.setExit("adventure", sBeach);

        //Food
        Food apple, toast, steak, magicCarrot, pear;

        apple = new Food(1, "red apple", 20);
        toast = new Food(1, "toast", 15);
        steak = new Food(1, "steak", 35);
        magicCarrot = new Food(2, "magic carrot", 100);
        pear = new Food(1, "sweet pear", 10);

        //Weapons
        Weapon sword, staff, beeGun, bow, axe;

        sword = new Weapon("Iron sword", 20, 0, 3, 0);
```

```
        staff = new Weapon("Thunder Staff", 80, 2, 7, 1);
        beeGun = new Weapon("Bee Gun", 75, 2, 7, 1);
        bow = new Weapon("Golden Bow", 35, 1, 5, 1);
        axe = new Weapon("Axe", 25, 0, 4, 0);

        //Trophies
        Trophy skull, feather, sting, chest;

        skull = new Trophy("Cannibal skull", 1);
        feather = new Trophy("Vulture feather", 1);
        sting = new Trophy("Bee sting", 1);
        chest = new Trophy("Treasure chest", 5);

        goal = chest;

        //Mobs
        Mob cannibal1, cannibal2, vulture, bee, captain;

        cannibal1 = new Mob("Cannibal", skull, 50, 5, 25, 70);
        cannibal2 = new Mob("Cannibal", skull, 50, 5, 25, 70);
        vulture = new Mob("Vulture", feather, 40, 10, 30, 65);
        bee = new Mob("Bee", sting, 20, 5, 12, 20);
        captain = new Mob("Captain Toad", chest, 200, 25, 80, 100);

        //Add items to rooms
        cabin.addWeapons(sword, 1);
        cabin.addWeapons(axe, 1);
        cabin.addWeapons(axe, 1);
        cabin.addFood(apple, 5);
        cabin.addFood(toast, 3);
        cabin.addFood(steak, 3);

        jungle.addFood(pear, 4);
        //jungle.addMob(cannibal1);
        //jungle.addMob(cannibal2);
        //jungle.addMob(bee);

        cave.addWeapons(beeGun, 1);

        lake.addFood(magicCarrot, 3);

        mountain.addMob(vulture);
        mountain.addWeapons(staff, 1);

        armory.addWeapons(bow, 7);
        armory.addWeapons(sword, 7);
        armory.addWeapons(axe, 7);

        chestRoom.addMob(captain);
        //====================
        map = new Map(mainLand, roof, cabin, teleport, allRooms, roomsToTeleportTo);
//Creating the actual map with mobs, rooms and items.
    }
```

```
}
```

```java
/**
 * Class Command - a command in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Command" represents a single valid command. After the user has input a line to
the terminal,
 * it is checked whether the input starts with a valid command.
 * After that appropriate actions are performed.
 *
 * @author Michael Kölling, David J. Barnes and Vakaris Paulavičius (Student number:
20062023).
 * @version 2020.11.26
 */
public class Command
{
    private String commandWord;
    private String secondWord;
    private String thirdWord;

    /**
     * Create a command object. First and second word must be supplied, but
     * either one (or both) 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.
     */
    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 second word of this command. Returns null if there was no
     * second 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
/**
 * Class TextCreator is used to create pseudo interface in the terminal.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * This class is used to print text to the terminal in various patterns,
 * to create a primitive interface for a better user experience.
 * Use of this class also prevents clutter in the game class.
 *
 * However, this class needs a lot of upgrading, because whenever a new command is
added,
 * all the patterns must be changed. In addition, if the text line is long enought,
it can go out of bounds
 * in the terminal.
 *
 * @author Vakaris Paulavičius (Student number: 20062023)
 * @version 2020.11.26
 */
public class TextCreator
{
    /**
     * Print out the opening message for the player.
     */
    public void printWelcome()
    {
        System.out.println();

System.out.println("------------------------------------------------------------------
-----------------------");
        System.out.println("*********************          Welcome to the
Peacemaker          *********************");

System.out.println("------------------------------------------------------------------
-----------------------");
        System.out.println("|             Bring the village peace by defeating the
brutal Captain Toad             |");

System.out.println("------------------------------------------------------------------
-----------------------");
        System.out.println();
    }

    /**
     * Print out the closing message for the player.
     */
    public void printGoodbye()
    {
        System.out.println("                        Thanks for your time, soldier. See
you soon!                      ");

System.out.println("------------------------------------------------------------------
-----------------------");
    }
```

```
    /**
     * Print top box.
     */
    public void printTopBox()
    {
        System.out.println("------------------------------------------------------------------------");
        System.out.println("|                    Type 'help' if you want to know available commands                 |");
        System.out.println("------------------------------------------------------------------------");

        System.out.println("|                                                                        |");
    }

    /**
     * Print bottom box.
     */
    public void printBottomBox()
    {
        System.out.println("|                                                                        |");

        System.out.println("------------------------------------------------------------------------");
    }

    /**
     * Print details about the current room.
     *
     * @param mobs A string with mobs that are in that room.
     * @param food A string with food products that are in that room.
     * @param weapons A string with weapons that are in the room.
     * @param trophies A string with trophies that are in this. room.
     */
    public void printCurrentRoomDetails(String mobs, String food, String weapons, String trophies)
    {
        System.out.println("------------------------------------------------------------------------");
        System.out.println(" M: " + mobs);
        System.out.println(" //");
        System.out.println(" F: " + food);
        System.out.println(" //");
        System.out.println(" W: " + weapons);
        System.out.println(" //");
        System.out.println(" T: " + trophies);
```

```
System.out.println("------------------------------------------------------------------
-----------------------");
        System.out.println();
    }

    /**
     * Print game win message.
     */
    public void winMessage(String string)
    {
        System.out.println();
        System.out.println("                          *** " + string + " has won! The
village celebrates.  ***");
        System.out.println();
    }

    /**
     * Print player creation text.
     */
    public void createCharacterText()
    {
        System.out.println("               Now it is time to create your own
character. Type 'female'  ");
        System.out.println("                    or 'male', to choose your character's
gender.          ");
        System.out.println("
OR                                ");
        System.out.println("                         Type 'quit' to exit this game
now.              ");
        System.out.println();
    }

    /**
     * Print text in a pattern specifically designed for the terminal interface.
     */
    public void classicPattern(String string)
    {
        System.out.println(" " + string);
        System.out.println();
    }

    /**
     * Print a single line. This method is just for consistency.
     */
    public void line(String string)
    {
        System.out.println(string);
    }

    /**
     * Print items that are currently in the player's backpack.
     *
     * @param food A string with food products that are in player's backpack.
```

```
     * @param weapons A string with weapons that are in the backpack.
     * @param trophies A string with trophies that are in the backpack.
     */
    public void backpackItems(String food, String weapons, String trophies)
    {
        System.out.println(" Your items: ");
        System.out.println("   Food: "+ food);
        System.out.println("   Weapons: "+ weapons);
        System.out.println("   Trophies: "+ trophies);
        System.out.println();
    }

    /**
     * Print text with character's name after user creates it.
     *
     * @param name Character's name.
     */
    public void characterText(String name)
    {
        System.out.println();
        System.out.println("                  Your character '" + name + "' was
successfully created!");
    }

    /**
     * Print text after completing the game, but without the main trophy.
     */
    public void noMainTrophyMessage()
    {
        System.out.println();
        System.out.println("              You lost, because you got out of the
dungeon without a chest.");
        System.out.println("                     Now the people of your village will
starve.");
    }

    /**
     * Print text when player enters the old man's cabin.
     */
    public void taskMessage()
    {

System.out.println("----------------------------------------------------------------
-----------------------");
        System.out.println(" The old man: ");
        System.out.println();
        System.out.println("* Hello warrior, nice to finally meet you. I am happy to
hear that there still are");
        System.out.println("  brave people who are not afraid to fight for good.
Captain Toad has been terrorizing");
        System.out.println("  our peaceful village for over a decade now.");
        System.out.println();
        System.out.println("* Your task is to travel to the North island and defeat
him. He usually hides in his");
```

```
        System.out.println("  dungeon.");
        System.out.println();
        System.out.println("* In this cabin, you can find some food and weapons.
However, to defeat this fiend,");
        System.out.println("  you'll have to find higher-level weapons and better
food.");
        System.out.println();
        System.out.println("* I suggest visiting the East and the West islands. There
you might find something,");
        System.out.println("  but be careful, these islands are full of dangerous
creatures.");
        System.out.println();
        System.out.println("* One final thing, after you defeat the captain, which I
am sure you will, take the");
        System.out.println("  chest with you before leaving. That money belongs to
the village people.");
        System.out.println("^ I wish you all the best!");

System.out.println("----------------------------------------------------------------
-----------------------");
        System.out.println();
    }

    /**
     * Print fight text in a specific pattern.
     *
     * @param playerDamage The amount of damage player's weapon does in one hit.
     * @param playerStats Player's health details.
     * @param mobHealth Health of the mob after player's hit.
     * @param mobDamage The amount of damage a mob does in one hit.
     * @param mobName Mob's name.
     * @param isAlive Current mob status: alive/dead.
     */
    public void fightText(int playerDamage, String playerStats, int mobHealth, int
mobDamage, String mobName, boolean mobAlive)
    {
        System.out.println(" ## ## ## ## ## ## ## ## ## ##");
        System.out.println(" You did " + playerDamage + " damage to the " + mobName
+ ".");
        System.out.println(" *    *    *");
        System.out.println(" " + mobName + " did " + mobDamage + " damage to you.");
        System.out.println(" //    //    //    //    //");
        System.out.println(" Your current health is: " + playerStats + ".");
        System.out.println(" *    *    *");
        if(mobAlive)
        {
            System.out.println(" " + mobName + " has " + mobHealth + " health
left.");
        }
        else
        {
            System.out.println(" You have slained the " + mobName + ".");
        }
        System.out.println(" ## ## ## ## ## ## ## ## ## ##");
```

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

    /**
     * Print text if player dies.
     */
    public void deathText()
    {
        System.out.println("                                    You died. Better luck next
time.");
    }

    /**
     * Print out available commands.
     *
     * @param commands A string with all valid commands.
     */
    public void printHelp(String commands)
    {
        System.out.println("   These are some useful commands that might help:");

System.out.println("-------------------------------------------------------------------
-----------------------");
        System.out.print("|  ");
        System.out.println(commands + " |");

System.out.println("-------------------------------------------------------------------
-----------------------");
        System.out.println();
    }
}
```

```java
import java.util.ArrayList;
import java.util.Random;
/**
 * Class Map - a map in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Map" represents a game map. It holds a list of rooms throughout which
 * a player can travel. An object of this class also has instance fields for
 * relevant rooms such as the start room, the finish room, the task room (where the
 * player is given a task in that map) and the teleport room (when a player enters
this
 * room, he is taken to another available random room).
 *
 * @author Vakaris Paulavičius (Student number: 20062023)
 * @version 2020.11.30
 */
public class Map
{
    // instance variables - replace the example below with your own
    private ArrayList<Room> availableTeleportDestinations;
    private ArrayList<Room> rooms;

    private Room startRoom;
    private Room endRoom;
    private Room taskRoom;
    private Room teleportRoom;

    private boolean taskRoomVisited;

    /**
     * Create a map and initialize the speacial rooms.
     */
    public Map(Room startRoom, Room endRoom, Room taskRoom, Room teleportRoom, Room[]
allRooms, Room[] teleportDestinations)
    {
        this.startRoom = startRoom;
        this.endRoom = endRoom;
        this.taskRoom = taskRoom;
        this.teleportRoom = teleportRoom;
        taskRoomVisited = false;
        availableTeleportDestinations = new ArrayList<>();
        rooms = new ArrayList<>();
        addRooms(allRooms);
        setTeleportDestinations(teleportDestinations);
    }

    /**
     * This method is used to get the start room of this map.
     * @return a Room which is the start room in this map.
     */
    public Room getStartRoom()
    {
```

```
    return startRoom;
}

/**
 * This method is used to get the end room of this map.
 * @return a Room which is the end room in this map.
 */
public Room getEndRoom()
{
    return endRoom;
}

/**
 * This method is used to get the teleport room of this map.
 * @return a Room which is the teleport room in this map.
 */
public Room getTeleportRoom()
{
    return teleportRoom;
}

/**
 * This method is used to get the task room of this map.
 * @return a Room which is the task room in this map.
 */
public Room getTaskRoom()
{
    return taskRoom;
}

/**
 * This method is used to add a single room to the map.
 *
 * @param room A Room which to add.
 */
public void addRoom(Room newRoom)
{
    rooms.add(newRoom);
}

/**
 * This method is used to add an array of rooms to the map.
 *
 * @param newRooms A Room array which objects to add.
 */
public void addRooms(Room[] newRooms)
{
    for(int i = 0 ; i < newRooms.length ; i ++)
    {
        rooms.add(newRooms[i]);
    }
}

/**
```

```
    * This method is used to add all the available locations to the array list
'availableTeleportDestinations'.
    *
    * @param rooms[] An array with Rooms available to teleport to.
    */
    public void setTeleportDestinations(Room rooms[])
    {
        for(int i = 0 ; i < rooms.length ; i ++)
        {
            availableTeleportDestinations.add(rooms[i]);
        }
    }

    /**
    * This method pick a random room from the list "availableTeleportDestinations"
and returns it.
    * @return a Room which the player teleport to.
    */
    public Room teleportToRoom()
    {
        Random rnd = new Random();
        return
availableTeleportDestinations.get(rnd.nextInt(availableTeleportDestinations.size()));
    }

    /**
    * This method returs if the taskRoom was visited.
    * @return true if it was, false otherwise.
    */
    public boolean taskRoomWasVisited()
    {
        return taskRoomVisited;
    }

    /**
    * This method is used to set the instance variable taskRoomVisited to true.
    */
    public void visitTaskRoom()
    {
        taskRoomVisited = true;
    }

    //Methods below were not used yet. For future updates.

    /**
    * This method is used to set the start room for this map.
    *
    * @param room A Room which is the start room in this map.
    */
    public void setStartRoom(Room room)
    {
        startRoom = room;
    }
```

```
    /**
     * This method is used to set the task room for this map.
     *
     * @param room A Room which is the task room in this map.
     */
    public void setTaskRoom(Room room)
    {
        taskRoom = room;
    }

    /**
     * This method is used to set the end room for this map.
     *
     * @param room A Room which is the end room in this map.
     */
    public void setEndRoom(Room room)
    {
        endRoom = room;
    }

    /**
     * This method is used to set the teleport room for this map.
     *
     * @param room A Room which is the teleport room in this map.
     */
    public void setTeleportRoom(Room room)
    {
        teleportRoom = room;
    }
}
```

```java
import java.util.Set;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Random;
/**
 * Class Room - a room in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Room" represents one single room in the game. Many rooms have their own mobs,
 * items and weapons.
 *
 * @author Michael Kölling, David J. Barnes and Vakaris Paulavičius (Student number:
20062023).
 * @version 2020.11.26
 */

public class Room
{
    private String description;

    private HashMap<String, Room> exits;        // stores exits of this room.

    private HashMap<Food, Integer> foodList;        //stores food that is in this
room
    private ArrayList<Mob> mobs;         //stores mobs that are in this room
    private HashMap<Weapon, Integer> weapons;            //stores weapons that are in
this room
    private HashMap<Trophy, Integer> trophies;          //stores trophies that are in
this room

    /**
     * Create a room.
     *
     * @param description Room's short description.
     */
    public Room(String description)
    {
        this.description = description;
        exits = new HashMap<>();
        foodList = new HashMap<>();
        mobs = new ArrayList<>();
        weapons = new HashMap<>();
        trophies = 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);
    }

    /**
     * This method is used to get room's short description.
     * @return short description of the room.
     */
    public String getShortDescription()
    {
        return description;
    }

    /**
     * This method is used to get a long description of the room in the form:
     *     You are in the kitchen.
     *     You can go: north west
     * @return a long description of this room
     */
    public String getLongDescription()
    {
        return description + "\n" + getExitString();
    }

    /**
     * This method is used to get a string describing the room's exits, for example
     * "You can go: north west".
     * @return Details of the room's exits.
     */
    private String getExitString()
    {
        String returnString = " You can go:";
        Set<String> keys = exits.keySet();
        for(String exit : keys) {
            returnString += " " + exit;
        }
        return returnString;
    }

    /**
     * This method is used to get 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.
     * @return The room in the given direction.
     */
    public Room getExit(String direction)
    {
        return exits.get(direction);
    }

    /**
     * This method is used to add food objects to the room.
```

```
     *
     * @param food Food object to add.
     * @param quantity How many food objects of this kind to add.
     */
    public void addFood(Food food, int quantity)
    {
        if(foodList.containsKey(food))
        {
            int amount = foodList.get(food);
            foodList.put(food, amount+quantity);
        }
        else
        {
            foodList.put(food, quantity);
        }
    }

    /**
     * This method is used to add weapons to the room.
     *
     * @param weapon Weapon object to add.
     * @param quantity How many weapons of this kind to add.
     */
    public void addWeapons(Weapon weapon, int quantity)
    {
        if(weapons.containsKey(weapon))
        {
            int amount = weapons.get(weapon);
            weapons.put(weapon, amount+quantity);
        }
        else
        {
            weapons.put(weapon, quantity);
        }
    }

    /**
     * This method is used to add mobs to the room.
     *
     * @param mob Mob object to add.
     */
    public void addMob(Mob mob)
    {
        mobs.add(mob);
    }

    /**
     * This method is used to add trophies to the room.
     *
     * @param trophy Trophy object to add.
     */
    public void addTrophies(Trophy trophy)
    {
        if(trophies.containsKey(trophy))
```

```
    {
        int amount = trophies.get(trophy);
        trophies.put(trophy, amount+1);
    }
    else
    {
        trophies.put(trophy, 1);
    }
}

/**
 * This method checks if there are any mobs in this room.
 * @return true if there are, false if there are not.
 */
public boolean hasMobs()
{
    if(mobs.size() > 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

/**
 * This method is used to remove food objects from the room.
 *
 * @param item Food object to remove.
 */
public void removeFood(Food item)
{
    if(foodList.get(item) == 1)
    {
        foodList.remove(item);
    }
    else
    {
        int quantity = foodList.get(item);
        foodList.put(item, quantity -1);
    }
}

/**
 * This method is used to remove trophies from the room.
 *
 * @param item Trophy object to remove.
 */
public void removeTrophy(Trophy item)
{
    if(trophies.get(item) == 1)
    {
        trophies.remove(item);
```

```
        }
        else
        {
            int quantity = trophies.get(item);
            trophies.put(item, quantity -1);
        }
    }

    /**
     * This method is used to remove weapons from the room.
     *
     * @param item Weapon object to remove.
     */
    public void removeWeapon(Weapon item)
    {
        if(weapons.get(item) == 1)
        {
            weapons.remove(item);
        }
        else
        {
            int quantity = weapons.get(item);
            weapons.put(item, quantity -1);
        }
    }

    /**
     * This method is used to remove mobs from the room.
     *
     * @param mob Mob to remove.
     */
    public void removeMob(Mob mob)
    {
        mobs.remove(mob);
    }

    /**
     * This method is used to get a food object from this room using its name.
     *
     * @param name Food object's name.
     * @return food object if it is in this room, null if the object was not found.
     */
    public Food getFoodByName(String name)
    {
        for(Food key : foodList.keySet())
        {
            if(key.getName().toLowerCase().equals(name)) return key;
        }
        return null;
    }

    /**
     * This method is used to get a mob from this room using its name.
     *
```

```
     * @param name Mob's name.
     * @return mob object if it is in this room, null if the object was not found.
     */
    public Mob getMobByName(String name)
    {
        for(Mob mob : mobs)
        {
            if(mob.getName().toLowerCase().equals(name)) return mob;
        }
        return null;
    }

    /**
     * This method is used to get a weapon from this room using its name.
     *
     * @param name Weapon's name.
     * @return weapon object if it is in this room, null if the object was not found.
     */
    public Weapon getWeaponByName(String name)
    {
        for(Weapon key : weapons.keySet())
        {
            if(key.getName().toLowerCase().equals(name)) return key;
        }
        return null;
    }

    /**
     * This method is used to get a trophy from this room using its name.
     *
     * @param name Trophy's name.
     * @return trophy object if it is in this room, null if the object was not found.
     */
    public Trophy getTrophyByName(String name)
    {
        for(Trophy key : trophies.keySet())
        {
            if(key.getName().toLowerCase().equals(name)) return key;
        }
        return null;
    }

    /**
     * This method is used to get all the mobs that are in this room.
     * @return a string with all the mobs that are in the room.
     */
    public String listMobs()
    {
        if(mobs.isEmpty())
        {
            return " There are no mobs in this place. We are safe, for now...";
        }
        else
        {
```

```
            String str = " " + randomNegativeResponse();
            //We put all the mobs into the hashmap to count their number using their
names.
            HashMap<String, Integer> mobCount = new HashMap<>();
            mobCount = countMobs(mobs);
            int index = 0;
            for(String key: mobCount.keySet())
            {
                if(mobs.size() == 1 && mobCount.get(key) == 1)
                {
                    str += " " + (mobCount.get(key) + " " + key + " is coming your
way!");
                }
                else
                {
                    if(index == (mobCount.size()-1))
                    {
                        if(mobCount.get(key) == 1)
                        {
                            str += " " + (mobCount.get(key) + " " + key + " are
coming your way!");
                        }
                        else
                        {
                            str += " " + (mobCount.get(key) + " " + key + "s are
coming your way!");
                        }
                    }
                    else
                    {
                        if(mobCount.get(key) == 1)
                        {
                            str += " " + (mobCount.get(key) + " " + key + ",");
                        }
                        else
                        {
                            str += " " + (mobCount.get(key) + " " + key + "s,");
                        }
                    }
                }
                index ++;
            }
            return str;
        }
    }

    /**
     * This method is used to get all the food items that are in this room.
     * @return a string with all the food products that are in the room.
     */
    public String listFood()
    {
        if(foodList.isEmpty())
        {
```

```
            return " There is no food in this place!";
        }
        else
        {
            String str = " " + randomPositiveResponse();

            int index = 0;
            for(Food key: foodList.keySet())
            {
                if(foodList.size() == 1 && foodList.get(key) == 1)
                {
                    str += " There is "+ (foodList.get(key) + " " + key.getName() + "
on the tree stump.");
                }
                else
                {
                    if(index == (foodList.size()-1))
                    {
                        if(foodList.get(key) == 1)
                        {
                            str += " " + (foodList.get(key) + " " + key.getName() + "
are on the tree stump.");
                        }
                        else
                        {
                            str += " " + (foodList.get(key) + " " + key.getName() +
"s are on the tree stump.");
                        }
                    }
                    else
                    {
                        if(foodList.get(key) == 1)
                        {
                            str += " " + (foodList.get(key) + " " + key.getName() +
",");
                        }
                        else
                        {
                            str += " " + (foodList.get(key) + " " + key.getName() +
"s,");
                        }
                    }
                }
                index ++;
            }
            return str;
        }
    }

    /**
     * This method is used to get all the weapons that are in this room.
     * @return a string with all the weapons that are in the room.
     */
    public String listWeapons()
```

```
    {
        if(weapons.isEmpty())
        {
            return " No weapons out here!";
        }
        else
        {
            String str = " " + randomPositiveResponse();

            int index = 0;
            for(Weapon key: weapons.keySet())
            {
                if(weapons.size() == 1 && weapons.get(key) == 1)
                {
                    str += " There is "+ (weapons.get(key) + " " + key.getName() + "
laying on the ground.");
                }
                else
                {
                    if(index == (weapons.size()-1))
                    {
                        if(weapons.get(key) == 1)
                        {
                            str += " " + (weapons.get(key) + " " + key.getName() + "
are laying on the ground.");
                        }
                        else
                        {
                            str += " " + (weapons.get(key) + " " + key.getName() + "s
are laying on the ground.");
                        }
                    }
                    else
                    {
                        if(weapons.get(key) == 1)
                        {
                            str += " " + (weapons.get(key) + " " + key.getName() +
",");

                        }
                        else
                        {
                            str += " " + (weapons.get(key) + " " + key.getName() +
"s,");

                        }
                    }
                }
                index ++;
            }
            return str;
        }
    }

    /**
     * This method is used to get all the trophies that are in this room.
```

```
     * @return a string with all the trophies that are in the room.
     */
    public String listTrophies()
    {
        if(trophies.isEmpty())
        {
            return " No trophies have been found in this location.";
        }
        else
        {
            String str = " " + randomPositiveResponse();

            int index = 0;
            for(Trophy key: trophies.keySet())
            {
                if(trophies.size() == 1 && trophies.get(key) == 1)
                {
                    str += " There is "+ (trophies.get(key) + " " + key.getName() + "
levitating in the air.");
                }
                else
                {
                    if(index == (trophies.size()-1))
                    {
                        if(trophies.get(key) == 1)
                        {
                            str += " " + (trophies.get(key) + " " + key.getName() + "
are levitating in the air.");
                        }
                        else
                        {
                            str += " " + (trophies.get(key) + " " + key.getName() +
"s are levitating in the air.");
                        }
                    }
                    else
                    {
                        if(trophies.get(key) == 1)
                        {
                            str += " " + (trophies.get(key) + " " + key.getName() +
",");
                        }
                        else
                        {
                            str += " " + (trophies.get(key) + " " + key.getName() +
"s,");
                        }
                    }
                }
                index ++;
            }
            return str;
        }
    }
```

```
    // ***** PRIVATE METHODS *****

    /**
     * This method is used to count the number of mobs. It takes an array list as a
parameter
     * and puts all the objects into the HasMap with their quantity as values.
     *
     * @param mobList A list of mobs that exist in that room
     * @return a HashMap with mob names and their quantity in this room. <String,
Integer>.
     */
    private HashMap countMobs(ArrayList<Mob> mobList)
    {
        HashMap<String,Integer> results = new HashMap<>();
        for(Mob mob : mobList)
        {
            if(results.containsKey(mob.getName()))
            {
                int quantity = results.get(mob.getName());    //If another mob of the
same name is found, the quantity is incremented by one.
                results.put(mob.getName(), quantity+1);
            }
            else
            {
                results.put(mob.getName(), 1);
            }
        }
        return results;
    }

    /**
     * This method is used to randomly select a positive response.
     * @return a positive response.
     */
    private String randomPositiveResponse()
    {
        Random rand = new Random();
        String choices[] = {"Superb!", "Fastastic!", "I think we can call it a
day.", "Is it Chistmas already?", "Look!", "Oh yes!",
                "Today is one lucky day for us, captain!", "I can't believe my
digital eyes!"};
        return choices[rand.nextInt(choices.length)];
    }

    /**
     * This method is used to randomly select a negative response.
     * @return a negative response.
     */
    private String randomNegativeResponse()
    {
        Random rand = new Random();
        String choices[] = {"Watch out!", "We've got a problem, captain!", "Oh no!",
"Be careful!", "Take a weapon!", "Take cover"};
```

```
        return choices[rand.nextInt(choices.length)];
    }
}
```

```java
/**
 * Class Food - a food product in the game.
 *
 * This class is part of "The peacemaker" application.
 * "The peacemaker" is a simple, text based adventure game.
 *
 * A "Food" represents one single food item in the game that can be used
 * to restore player's health. Game characters can have, eat and pickup
 * food products from different locations.
 *
 * @author Vakaris Paulavičius (Student number: 20062023)
 * @version 2020.11.26
 */
public class Food
{
    private int size;
    private int value;
    private String name;
    private String description; //Description of this food product

    /**
     * Create a new food object.
     *
     * @param size How much space units this food product covers.
     * @param name Name of this food product.
     * @param value How much health it restores.
     */
    public Food(int size, String name, int value)
    {
        this.size = size;
        this.name = name;
        this.value = value;
        setDescription();
    }

    /**
     * This method is used to get the size of this food product.
     * @return how much space units it covers.
     */
    public int getSize()
    {
        return size;
    }

    /**
     * This method is used to get the name of this food.
     * @return food product name.
     */
    public String getName()
    {
        return name;
    }

    /**
```

```
    * This method is used to get the amount of health this food product restores.
    * @return how much health this object restores
    */
    public int getValue()
    {
        return value;
    }

    /**
    * This method is used to get the description of the food object
    * @return food product description
    */
    public String getDescription()
    {
        return description;
    }

    // ***** PRIVATE METHODS *****

    /**
    * This method is used to set a new description for the food object
    */
    private void setDescription()
    {
        String line1 = ("Name: " + name),
               line2 = ("Size in backpack: " + size),
               line3 = ("Restores health: " + value);

        description = line1 + "\n " + line2 + "\n " + line3;
    }
}
```

```
* Project: "The Peacemaker"
* Authors: Michael Kölling, David J. Barnes and Vakaris Paulavičius (Student number:
20062023).
* Version: 1.1
* Date:    2020.11.28

* "The peacemaker" is a simple text-based adventure game.
  Here a player travels throughout a fictitious map with a goal to defeat the the
main boss
  and bring the peace to the village as well as a chest full of coins.

* To start this application, call the main class method of the "Game" class and enter
no parameters.
  In the beginning, create your own character and proceed to the journey.

* To win the game, a player must go "out" of the boss room with a chest in his
backpack.

* All the relevant details are explained when a player enters the old man's cabin. It
can be found in the South island.

* There is a primitive map among the project files which will hopefully help the user
to imagine the game's world more easily.
  It is called: "mapV1.png";

* Existing islands: the North island (Ruthlessness island)  //where the boss is
                    the South island (Placidity island) //where the old man lives
                    the West island (Entanglement island) //jungle, cave, teleport
                    the East island (Elevation island) //lake, mountain

* Purpose of a command:

       go:     Used to go to the next available place.
       quit:   Used to quit the game.
       help:   Used to get all available commands.
       hit:    Used to hit a mob.
       eat:    Used to consume a food product and increment health.
       look:   Used to look around the current place and get information about items
that are in that place.
       take:   Used to pick up an item from the player's current location.
       drop:   Used to drop an item from the backpack at the player's current
location.
       info:   Used to get the name of the current place and all available exits.
       list;   Used to get a list of items that are in the backpack.
       stats:  Used to get health and backpack stats.
       equip:  Used to equip a weapon that is in the backpack.
       about   Used to get a description of an item that is in the backpack or in the
current room. It can also be used to get a description of a mob that is currently in
that room.
       show:   Used to get information about the weapon that is currently equiped.

* How to use a command:
       go:     > go [place] (exactly as it is written);
       quit:   > quit;
```

```
help:   > help;
hit:    > hit [mob's name];
eat:    > eat [food item's name];
look:   > look;
take  : > pickup [item's name];
drop:   > drop [item's name];
info:   > info;
list;   > list;
stats:  > stats;
equip:  > equip [item's name];
about   > about [item's name];
show:   > show;
```