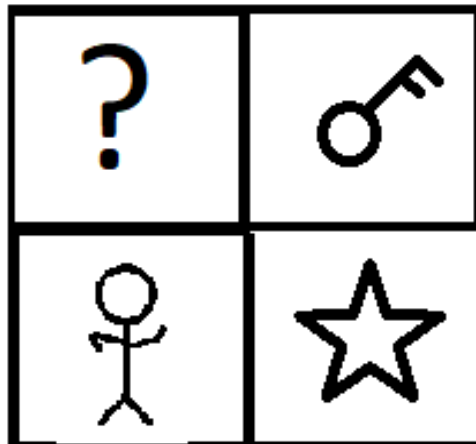


World Navigator Assignment



Done by :Aya Aridah
Supervised by: Dr.Motasem Diab

Acknowledgment

I would like to express my special thanks to Dr.Motasem and atypon for this golden internship opportunity that helps me a lot to discover so many fields I didn't know about it, thanks for helping us on projects, tasks and give an honest feedback about our progress which helps me a lot to know my programming level.

Thanks for enlighten us more than ever, that university courses are level -5 out of level 100 of the programming world.

Thanks for giving us the amazing resources that helps me a lot knowing from where to start and how. I refactored the code multiple times and each time I went through setbacks, I went and read every chunk of that related error that occurs. And finally, I enjoyed through this project, because some nights I was crying to find the solution and from where to start but that changed me, how I think and how to organize the code for such a big project like this.

Table of Contents

Clean Code Principles	4
Effective Java Code Principles	8
Design Patterns	13
SOLID Principles	15
Data Structures	17
Google styling guide.....	18

Clean Code Principles:

- **Use Intention-Revealing Names**

No abbreviations were added to the project to achieve the highest understanding. For example, the `Lockable` interface the name implies that the object can be locked or unlocked, and it's applied to the door and the chest.

```
public interface Lockable {  
    public String useKey(Key key);  
}
```

And each method has a clear name defining what it does like `useKey` method, The naming of methods were almost as recommended by starting with a verb then a noun. Also, `Buyable` interface implies that the object can be bought like flashlight and some specifically named keys, and it has a clear method name `buyItem` that tells its functionality from its name.

```
public interface Buyable {  
    public void buyItem(interfaces.Containable item);  
}
```

And here are some of methods from `Engine` class that represents using proper names:

```
private char getMainCommand()  
private char getPaintingCommand()  
private char getMirrorCommand()  
private void runBuyProccess(Tradable seller, ArrayList<Containable>  
sellerItems)  
private void diplayBaseCommands()
```

Boolean variable naming or methods always starts with `is`; you can find it for example at `Room` class like, `isDark` method:

```
public boolean isDark(FlashLight fl) {  
    if (fl.isOn()) {  
        return false;  
    }  
    return isDark;  
}
```

- **Use Solution Domain Names**

`DoorBuilder` makes it clear that the class represents builder design pattern.

`SingletonEngine` makes it clear that the class represents singletone design pattern.

```
public class DoorBuilder  
public class SingletonEngine  
public interface Facade  
public class RoomBuilder
```

- **Hungarian Notation**

Following the book's advice that every object used **MUST** be renamed including the form like if the variable is boolean you put the first letter is b (I saw many examples when use GUI), for example:

```
private boolean bisDark = false;
private boolean bhasLights = true;
```

- **Functions Should Be Small**

Following the book's advice that all functions prefer to be no more than 20 lines give or take. At Engine class the `start` method is broken down to a smaller methods like `getMainCommand()` and `showMainOptions()` each one has a single responsibility, for example:

```
public void start() {
    System.out.println("Hello! Welcome to the game..\n Good Luck :)
\n");
    gameTimer = new MyTimer(this.timeForGame);
    showMainOptions();
    while (true) {
        char commandEntered = getMainCommand();
        while (runNavCommand(commandEntered)) {
            commandEntered = getMainCommand();
        }
        runBaseCommand(commandEntered);
    }
}
```

In This way the methods are small, clear and readable.

- **Functions Should Do One Thing**

Functions should do one thing. They should do it well. They should do it only. The `getSellerCommand()` method is responsible for getting only seller command from the user. In This way the methods are small, and do one thing only!.

```
private char getSellerCommand() {
    char commandCharecter;
    while (true) {
        commandCharecter = getCharacter();

        for (SellerCommands c : SellerCommands.values())
            if (c.asChar == commandCharecter) return commandCharecter;
        if (commandCharecter == 'b') return commandCharecter;
        System.out.println("** command character entered not from the
options **");
    }
}
```

- **Don't Repeat Yourself (DRY principle)**

Instead of repeating the code in the `look` method for the `Door`, `Chest`, `Mirror`, `Painting`, `Plain`, And for the `Seller` class. I made a method for handling these classes using the fact that the implementation depends on the `Wallable` interface, and the same for the `Chest` and the `Door` class as the implementation depends on the `Openable` interface.

```
public interface Wallable {
    public String look();
}
public interface Openable {
    public void open();
}
```

- **Commented Out Code**

The problem with commented out code is that it can hide what you're actually trying to say in a wall of text. I didn't use any comments in the code and when I have the situation to write a comment I took my time thinking if any comment here could make the code more readable. But as the book said I made a good comment for separating the methods of show commands and the methods in backend that responsible for logic of the game.

```
private boolean isLinkingEndRoom(Door door) {
    if (door.getSideRoom() == this.gameMap.getEndRoom()) return true;
    else return false;
}

// print Frontend commands

private void showDoorSubOptions() {
    for (int i = 0; i < DoorCommands.values().length; i++)
        System.out.println(
            "    " + DoorCommands.values()[i].asChar + " -> " +
            DoorCommands.values()[i]);
    System.out.println("\n    b -> return to main menu");
}
```

- **The Boy Scout Rule**

The Boy Scout Rule can be summarized as: **Leave your code better than you found it.** Paying attention to refactoring is necessary in order to keep the code in a state where it is able to extend and maintain it for the future. And I implement the classes and interfaces in a way for future when extend, for example: MVC design pattern can be added without harming the project, because of including classes into packages. Also, the code inside could be extended and maintaining easily.

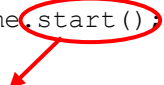
- **Vertical Openness between Concepts**

Making a blank line between concepts; like between methods or import statements and class declaration. And this advice is also enforced by the Style Guide.

- **Dependent Functions**

Dependent methods should be close together. Like, if we have a caller method and callee method they should be closer enough, if at all possible. For example, `init` method (the caller for `start` method) is on top of its the callee (`start` method) and so on the rest of the project.

```
private void init() {  
    Engine restartedGame = GameFileBuilder.loadGameInfo(this.gameName);  
    restartedGame.start();  
}  
  
public void start() {  
    System.out.println("Hello! Welcome to the game..\n Good Luck :)  
    \n");  
    gameTimer = new MyTimer(this.timeForGame);  
    showMainOptions();  
    while (true) {  
        char commandEntered = getMainCommand();  
        while (runNavCommand(commandEntered)) {  
            commandEntered = getMainCommand();  
        }  
        runBaseCommand(commandEntered);  
    }  
}
```



- **Horizontal alignment**

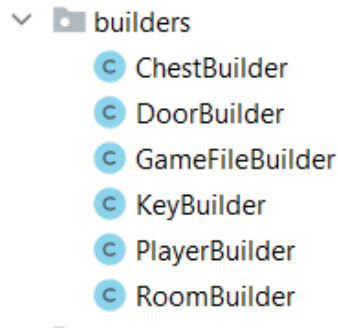
I didn't use this as it's not required and in the book stated that it's not a good idea.

The alignment seems to emphasize the wrong things and leads the eye away from the true intent.

Effective Java Code Principles:

- **ITEM 2: Consider a Builder When Faced With Many Constructor Parameters**

I used `ChestBuilder`, `DoorBuilder`, and a `KeyBuilder` and many more builders to help for their creation. It's really useful that you can see the name of each parameter you sent, and it's really good that you can easily ignore other parameters.



For example, In `DoorBuilder`, we can make the door open or closed, you can also specify the name or leave it as default, you will set it anyway later maybe.

```
public class DoorBuilder {  
  
    private String name;  
    private Key requestedKey ;  
    private boolean isOpen = true;  
  
    public DoorBuilder setName(String name) {  
        this.name = name;  
        return this;  
    }  
  
    public DoorBuilder setRequestedKey(Key requestedKey) {  
        this.requestedKey = requestedKey;  
        return this;  
    }  
  
    public DoorBuilder setOpen(boolean open) {  
        this.isOpen = open;  
        return this;  
    }  
  
    public Door build() {  
        if ((this.requestedKey.getDescription() == "Null") || this.name == null)  
            throw new IllegalArgumentException();  
  
        return new Door(this.name, this.isOpen, this.requestedKey);  
    }  
}
```


- **ITEM 3: Enforce the Singleton Property with a Private Constructor or an Enum Type**

The Singleton's purpose is to control object creation, while making sure that only single object from Engine class gets created. So, we make the `demoGame` member variable private and constructor as private. A public static factory method is used to get reference to the object.

```
public class SingletonEngine {  
  
    private static Engine demoGame;  
    private SingletonEngine() {  
    }  
  
    public static Engine getControls() {  
        if (demoGame == null) {  
            GameMap gameMap = new MapCreation().getDemoMap();  
            demoGame = Engine.create("newGame", gameMap, new  
PlayerBuilder().build(), 600000);  
        }  
        return demoGame;  
    }  
  
}
```

- **ITEM 5: Prefer Dependency Injection to Hardwiring Resources**

All the static functions in the classes does not need any dependency from that class, mainly used to implement something like class Name which helped the other classes, or is in classes that really can work fully static, like the GameFileBuilder which is responsible for serializing and deserializing the games to files and saving them.

- **ITEM 6: Avoid Creating Unnecessary Objects**

In many of the classes especially the Engine, if there is an object that can be used in many methods, we take this common use in one method to prevent the cost of new objects. `getNumberInRange` method is responsible for all the Integer inputs in that class. Giving that, almost every method needs Scanners in a way of another. But, with good organization will be no unnecessary objects!

```
private int getNumberInRange(int from, int to) {  
    Scanner in = new Scanner(System.in);  
    int commandNumber;  
    boolean first = true;  
    do {  
        if (!first) {  
            System.out.println("** number not in the range **");  
        }  
        while (!in.hasNextInt()) {  
            String input = in.next();  
            System.out.printf("\n%s\n is not a valid number.\n",  
input);  
        }  
    } while (true);  
    return commandNumber;  
}
```

- **ITEM 9: Prefer Try-With-Resources to Try-Finally**

Try-With-Resources is used by the `GameFileBuilder` class when loading the game file. And as the book stated it's shorter, more readable, better handles closing resources properly and provides far better diagnostics even in the face of an exception or return.

```
public static void serializeGameInFile(Engine engine, String filePath)
{
    try {
        FileOutputStream fileOut = new FileOutputStream(filePath);
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(engine);
        out.close();
        fileOut.close();
    } catch (IOException i) {
        i.printStackTrace();
    }
}
```

- **ITEM 12: Always override toString**

As the word (Always) means, every class in the project override the `toString()` method, just one simple try to use the base `toString()` to make the items have a proper string representation, any time that user will use `toString` he can see the base information that I provided rather than the class name with its memory address. For example: in class `Door`

```
@Override
public String toString() {
    return this.name + " Door";
}
```

- **ITEM 15: Minimize the Accessibility of Classes and Members**

In the program, most of the times you can't access anything except the interface methods; the program will cast the object to it and use your public implementation of that method. `(Checkable)(item).check();` All Instance fields of public classes are private in the program..

- **ITEM 17: Minimize Mutability**

Some immutable classes in the project are `Key`, `Wall` and `Seller`. In the `Key` class the key price is unmodifiable, so it's fine to return access to it by the `getPrice` method.

```
@Override
public int getPrice() {
    return price;
}
```

- **ITEM 20: Prefer Interfaces to Abstract Classes**

I didn't use any abstract class in the whole design, instead I used interfaces .Sometimes it may be hard to use interfaces in the cases that you have common behavior, but we would rather using the interface and handle that part alone, in interfaces, nothing set before.. just the method you need and you can implemented by your way.

- **ITEM 22: Use Interfaces Only to Define Types**

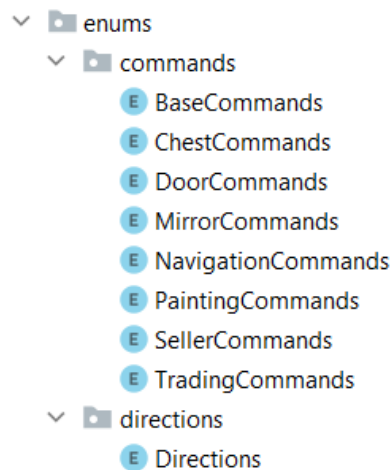
Just to mention that there is no interfaces that contains any final value or we deal with it wrongly be saving data in it.

- **ITEM 28: Prefer Lists to Arrays**

We all know that arrays are good in general, but the idea of having **static** size is scary! In this program we use ArrayLists most of the times, extendable with the ability to control it..

- **ITEM 34: Use Enums Instead of Int Constants**

9 Enums used (at this time) in the program, it's very helpful especially if you want to set some details about your constant value!



- **ITEM 50: Make Defensive Copies When Needed**

In this program I apply this in a different way, if you have the options to restart the game, that means that you need your initial values back ,but because of the problem of referencing, I made a copy file that will be used again to initialize again safely.

```
public static Engine create(String gameName, GameMap map, Player player,
int timeForGame) {
    Engine newGame = new Engine(gameName, map, player, timeForGame);
    GameFileBuilder.saveGameInfo(newGame);
    return newGame;
}
```

the problem of other solutions is that most of them required public constructor to create different instance, but since why have this option we used it

- **ITEM 51: Design Method Signature Carefully**

Method names are picked carefully, based on Google styling guide and relative names that shows what is the main function of the method.

- **ITEM 64: Refer to Objects By Their Interfaces**

while I was implementing an interface, and we are using that object in a situation that uses the interface, the interface is used to refer always (All the design works with interface references).

```
if (subCommand == DoorCommands.check.asChar) {
    if (((CheckableForOpenability)

this.currentRoom.getWallInDirection(this.player.getDirection()).getWallCo
ntent())
        .check()) {
        System.out.println("Door is open");
        if (isLinkingEndRoom(
            (Door)

this.currentRoom.getWallInDirection(this.player.getDirection()).getWallCo
ntent()))
            wonGame();
    }
```

in this example you can see how we really doesn't need to refer to the containing object specifically.. because we are in a check command process just referred to **CheckableForOpenability**

- **ITEM 77: DON'T IGNORE EXCEPTIONS**

In the program, all the Exceptions used in STANDARD ones(They achieve the job!). No exception is ignored, some of them send a message, and some others send where exactly the error is.

Design Patterns

1. Singleton Design Pattern

To implement a Singleton pattern, we have different approaches but all of them have the following common concepts. I chose the lazy Initialization and it's to implement Singleton pattern creates the instance in the global access method. But, the following implementation works fine in case of the single-threaded environment. And the reason of choosing singleton to use it into Engine class that there was only one controller that was controlling whole game so here singleton pattern come handy as it created one object that is used to control whole system rather than creating multiple object that perform same thing.

```
public class SingletonEngine {  
  
    private static Engine demoGame;  
    private SingletonEngine() {  
    }  
  
    public static Engine getControls() {  
        if (demoGame == null) {  
            GameMap gameMap = new MapCreation().getDemoMap();  
            demoGame = Engine.create("newGame", gameMap, new  
PlayerBuilder().build(), 600000);  
        }  
        return demoGame;  
    }  
}
```

2. Builder Design Pattern

A Builder class builds the final object step by step. Using Builder design pattern is a good strategy as it create object of all Class that are used in it deserializing and serializing object, made the code neat and easily understandable.

My problem at the past (before knowing builder design pattern) is that the arguments passed into a certain constractor when creating an object is the order of the arguments and it must be filled all with values, what if I don't want to assign a value for some certain object, here the builder patterns will be useful.

Once I read about this pattern I loved it very much so, I decided to use it to build each element in my game so I created ChestBuilder, DoorBuilder, KeyBuilder, PlayerBuilder and a RoomBuilder.

In MapCreation class I used the whole builders to build the map by building the rooms, the walls, the doors and whatever you want to build the room using the roomcomponents package. For example, this is a part of how I built the Sky Room in MapCreation class.

```

Room skyRoom = roomBuilder.setDark(false).setHasLights(true).build();
Seller sellerInSkyRoom = new Seller();
Key skyKey = this.keyBuilder.setName("Sky").setPrice(50).build();
Door skyDoor =
this.doorBuilder.setName("Sky").setOpen(false).setRequestedKey(skyKey).build();
skyKey.setRelatedLock(skyDoor.getName());
this.itemsInMap.add(skyKey);
Painting paintingSkyRoom = Painting.valueOf(skyKey);
skyRoom.getWallInDirection(Directions.EAST).setWallContent(paintingSkyRoom);
skyRoom.getWallInDirection(Directions.WEST).setWallContent(sellerInSkyRoom);

```

3. Façade Design Pattern

Facade is used to hide implementation of classes in package as classes of nullObjects only returns Null as a name so I have used Facade to hide implementation, we can get Name of classes through Facade interface.

```

public interface Facade {
    String getClassName();
}

```

4. Decorator Design Pattern

I was at the beginning of implementing this pattern, but after thinking a lot I decided not to use it because this pattern allows us to decorate the wall of multiple elements, for example on the same wall maybe it could be a seller and a painting but, the game doesn't requires us to make the wall contains multiple elements, so I decided not to implement it to focus more on clean code and other principles and managing time between learn all the resources we've got. I guess that it will be good if we want to extend the game at the future.

SOLID Principles

1. Single Responsibility Principle (SRP):

At the beginning, the whole picture of the project and design will be clearer when you really care about this principle, and that's what happened. In the project specifically at Engine class it's more important to try to distinguish between everything and give it its own responsibility, it is preferred to make every object alone, less coupling for sure and not that much care about memory (in the trade with performance).

Each class in the project is a standalone entity; you can actually use it in different situations and projects.

For example: In the Project you can find that the Wall with his ability to contain objects on it, makes it really challenging but with the help of Interfaces made it more flexibility. The Wall class can **contain** object, not every object but mainly what the player can **look** at it. So, the wall is an object which can be plain, mirror, seller or even a chest (or connected to it). And that is exactly how the Wall class built!

Wallable interface, is an interface that describes what we need in our game to be contained in the Wall, it includes **look()** method. And **look()** method has a single responsibility which is to look what the wall contain.

Note: I made it firstly, that the wall could contain many items on different positions, using decorator design pattern but in our game, the wall could contain one item at a time, but for the future, to extend the game we can implement it easily.

Coupling: unfortunately coupling appears in the project at MapCreation class which prepares a map including its rooms, then each room prepares its walls, then each wall prepares its items (Door, Chest, Mirror, Painting or Seller). I made my best thinking of this chain but I lastly found that this is a normal correct chain, where there is no map without a room at least, and a room without 4 walls as the requirements said, and walls may have items and each item has a builder that can be updated if any new item is to be added later.

2. Open Closed Principle (OCP):

In my project if we thought one day about adding something new, it can be on a Wall, and maybe contain something, you can really add it with 2 moves, implement **lookable** and **Checkable** interfaces, and the rest of the program will understand everything about your new item without any touch, because if they want to see what's inside you, check method will give them everything that they want..

What if I want to add a lock on this new item, I can't really act with the others since I added 2 separate behaviors that will affect each others, because if it's locked I can't get what inside! In this case you have many solutions actually not just one. You have sub-Interfaces in the

Checkable interface that can help you, and you have Lockable interface also to help if your behavior is really different and may relate to others.

3. Liskov Substitution Principle (LSP):

Honestly, this principle was helpful which used in my implementation of this project in one of the main scenarios; when having an Item class that is extended to all items such as door, chest, mirror, and painting but still, each one has its own specific fundamentals so doors are checkable but for operability, but the chest is Checkable but ForLockedContent, and the mirror is checkable but not openable. So having this principle to implement this critical scenario was helpful. So, I made CheckableForLockedContent, CheckableForContent and ChecakbleForOpenability interface to deal with all possible scenarios with all the items and any future added items.

4. Interface Segregation Principle (ISP):

You probably can see that all of the interfaces actually have one method, but considering Tradable, you will see three, and when I was thinking about it, when you say trade this will indicate both! Buying and selling!

Firstly, I made the interface like this `public interface Tradable { public ArrayList getItems(); public void sellItme(); public void butItem(); }` And as you can see there is a problem here, imagine if we have in the future a class that have the ability to buy items? If we implemented the Trade we will have to deal with getItem and sellItem methods!

So after thinking, why not to make each one of the behaviors as a standalone one? and we can make the interface extends another in the case of needing it! see the final decision .. `public interface Tradable extends Buyable, Sellable { public ArrayList getItems(); }` And as you can see, Buyable and Sellable contains **one** method representing their behavior.

5. Dependency Inversion Principle (DIP):

Almost all the classes that may contact with each other is communication with an general interface that will be implemented in each class in a different way (Maybe not and then we can consider strategy patterns) .

Not just interfaces, any kind of separation is consider in this principle, if you are having a class that contains many different objects you have to make sure that they doesn't get linked or connected in a way or another.

Player Class in the project is having different types of items inside it, but **Containable** interface took care of that, it has the functionalities that they need to use inside the Player class and nothing else.

Data Structures

Data Structures used in the Project is simple, Objects will include themselves, and when we are dealing with the case of having more than one object saved, I used **Lists**.

Arrays is better and simpler, but as **Effective Java** tells, and as the experiences that you went throw tells there is really many situations that the array will be very un useful. But, in class **Room** the Data structure that saves that values for the walls in the room is **Array** with default size **numberOfRooms** which is something have to be specified in our game. If we want to add walls you will simply change the number of walls in the room, add the new directions in the Enum, and everything will be good.

```
public class Room implements Lightable, Serializable {

    private final int wallsNumber = 4;

    private boolean isDark;
    private boolean hasLights;
    private Wall[] roomWalls;

    public Room(boolean isDark, boolean hasLights) {
        this.isDark = isDark;
        this.hasLights = hasLights;

        this.roomWalls = new Wall[wallsNumber];
        for (int i = 0; i < this.wallsNumber; i++) this.roomWalls[i] = new Wall();
    }
}
```

On the other side, that may be the only use of Arrays in the project, **ArrayList** is used many times to store the objects in it with the ability to shrink easily, as is Chest content and many others.

HashMap is also used, when the case of having non integer based parameter like in Chest check method, it is used as a simple Mapping implementation of Map in java, initial capacity given is (1), because it is used mostly in that size, but as you know, if there is a need for more, it will automatically response..

```
public HashMap<Boolean, ArrayList<Containable>> check() {
    if( this.isOpen()){
        ArrayList<Containable> temp = this.chestContents;
        this.chestContents = new ArrayList<Containable>();
        HashMap<Boolean,ArrayList<Containable>> res = new HashMap<>(1);
        res.put(true,temp);
        return res;
    }
    else {
        ArrayList<Containable> keyHolder = new ArrayList<>();
        keyHolder.add(this.requiredKey);
        HashMap<Boolean,ArrayList<Containable>> res = new HashMap<>(1);
        res.put(false,keyHolder);
        return res;
    }
}
```

Google styling guide

- **Source file basics**
 - **Special escape sequences**

For any character that has a special escape sequence (\b, \t, \n, \f, \r, \", \' and \\), that sequence is used rather than the corresponding octal (at Engine class – runBaseCommand method) and all the project uses the escape sequence not the octal ones.

```
if (mainCommand == BaseCommands.Restart.asChar) {
    System.out.println("Restarting ..\n \n \n");
    this.init();
    start();
}
```

- **Source file structure**

A source file consists of, **in order**:

1. License or copyright information, if present
2. Package statement
3. Import statements
4. Exactly one top-level class

Exactly one blank line separates each section that is present. Example from ChestBuiler:

```
package builders;

import interfaces.Containable;
import roomcomponents.Chest;
import roomcomponents.Key;
import java.util.ArrayList;

public class ChestBuilder {
```

- **Formatting**
 - **Braces are used where optional**

Braces are used with if, else, for, do and while statements, even when the body is empty or contains only a single statement.

```
public Chest build() {
    if ((this.requiredKey.getDescription() == "Null") || this.chestName == null)
        throw new IllegalArgumentException();
    return new Chest(this.chestName, this.isOpen, this.requiredKey, this.content);
}
```

- **Nonempty blocks: K & R style**

1. No line break before the opening brace.
2. Line break after the opening brace.
3. Line break before the closing brace.
4. Line break after the closing brace, only if that brace terminates a statement or terminates the body of a method, constructor, or named class. For example, there is no line break after the brace if it is followed by else or a comma.

```
public void switchLights() {  
    if (this.hasLights) {  
        this.isDark = !isDark;  
        System.out.println("lights switched to " + ((isDark == false) ?  
"on" : "off"));  
    } else {  
        System.out.println(Room.className() + " doesn't have lights.");  
    }  
}
```

- **One statement per line**

I tried to apply this formatting in all project

- **Column limit: 100**

I tried to apply this formatting in all project, and here an example that explain this point and the previous one

```
otherSideDoor.setSideRoom(this);  
otherSideDoor.setLinkedDoor(door);  
room.addDoorFrom(otherSideDoor, this, Directions.getOppositeDirection(direction));
```

Sorry for the miss (the indentation of the box), just to show the full sentence and it is 87 chars and it's the longest statement except package and import statements

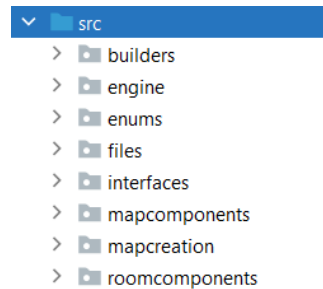
- **One variable per declaration**

```
private boolean isDark;  
private boolean hasLights;  
private Wall[] roomWalls;
```

- **Naming**

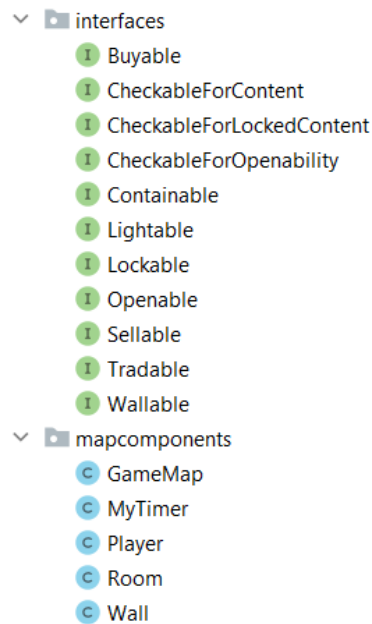
- **Package names**

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example



- **Class names**

Class names are written in UpperCamelCase. Class names are typically nouns or noun phrases. Interface names may also be nouns or noun phrases, but may sometimes be adjectives or adjective phrases instead.



• Programming Practices

- **@Override**: always used

```
@Override
public String look() { return Door.className(); }

@Override
public boolean check() { return isOpen; }

@Override
public void open() {
    if(isOpen)
        System.out.println("nothing happens\n");
    else
        System.out.println(this.requestedKey.getName()+" key required to unlock\n");
}

@Override
public String useKey(roomcomponents.Key key) {
    Objects.requireNonNull(key);
    if (this.requestedKey == key) {
        isOpen = !isOpen;
        this.linkedDoor.setOpen(isOpen);
        return(Door.className()+ ((isOpen == false) ? " looked" : " opened"));
    } else
        return(key.getName() + " key is not suitable for this door." + this.requestedKey.
}
```