The Maze Game

Done by Baha Gdiesat

Table of contents

ABSTRACT

The maze game:

Is an adventure-puzzle game where the main challenge is to solve a maze made of rooms. Each room consists of 4 walls. each Wall may have one of the following:

-Chest -Painting -Mirror -Door -Seller.

"graphic adventure: In adventure games, players usually interact with their environment and other characters to solve puzzles with clues to progress the gameplay. Aside from an occasional mini-game, adventure games rarely involve any traditional video game action elements. Thus, the genre isn't very popular with gamers.

Puzzle logic: Puzzle or logic games usually take place on a single screen or playfield and require the player to solve a problem to advance the action. A logic game requires players to solve a logic puzzle or navigate a challenge like a maze."

The maze game can be played online single player or multiplayer (with a group of people).

Each of the players can act on the maze using console commands or keyboard buttons or using the GUI in the web.

Game rules

1. The Players can start with initial amount of Gold.
   -to ensure fairness all the players must start with same amount of Gold-
2. If a Player exits one of the special doors representing the maze exit point, he/she wins while the other players lose the game (the game has only one winner).
3. If the maze timer elapses without exiting the maze it is a loss to all the Players.
4. all players must start the game at the same time and play the game simultaneously.
5. During the game, if two players happen to meet, they must fight, the loser immediately exits the game while the winner obtains all the items from the loser and the loser gold will be distributed among all the players.
6. in any fight, the winner is the player who have more gold (substitute 10 gold coins for each key and 2 gold coins for each flashlight). In case of tie, the winner will be selected randomly.
7. A player may give up and exit the game anytime. In such a case, the player's gold is distributed between all players. All items will be put in a chest (or a painting or a mirror) in the room, where the player gave up the game. Another optional feature is "drop on the floor" feature, in which a player who wants to give up throw all his/her items on the floor. Afterwards, another player who happened to walk in into this room will automatically obtain all the items on the floor.

The Game Commands

1) left, right: changes the orientation of the player.

2) forward, backward: will only move through open doors.

3) playerstatus: gives which direction the player is facing, amount of gold and items he has.

4) look: returns one of the following:
  a) dark: if the room is dark.
  b) door: if the player is facing a door.
  c) wall: if the player is facing a wall.
  d) chest: if the player is facing a chest.
  e) painting: if the player is facing a painting.
  f) seller: if the player is facing a seller.
  g) you see a silhouette of you: if the player is facing a mirror.

5) check: this command takes these arguments:
  a) mirror: if a key hidden behind the mirror will acquire the named key.
  b) painting: if a key hidden behind the painting will acquire the named key.
  c) door: if the door is locked "door is locked keyName is needed", and if the door is open "door is open".
  d) chest: will work only if facing a chest, if the chest is closed "chest closed <name> key is needed to unlock", if it is open the items inside the chest are listed and looted/acquired and could contain (named key, flashlight, or N gold where N is either a random number or a specific number set by the map).

6) Open, will open a door if the player is facing a door and it is unlocked and if the door is open "nothing happens" or if locked "<named> key required to unlock".

7) Trade, will work only if the player is facing a seller, will list the seller's available items (including possibly a flashlight and specifically named keys with prices in gold, the items of the seller are specified by themap).

**The subcommands of Trade are:
■ Buy <item>: if enough gold is with the player will "<item> bought and acquired" otherwise, the seller will say "return when you have enough gold".
■ Sell <Item>: The seller will have a price list of any item type that can be on the map and will offer that amount for any item that you have. You can buy back items at the same price.

■ List: will list seller items again.
■ Finish Trade: will exit the trade mode.

8) Use flashlight: will turn it on if it is off and vice versa and a dark room will become lit if the flashlight is on.

9) Use <name> Key: will open if a door/chest requires the <name> key to open and it is locked and will lock the door/chest if it is open.

10) SwitchLights: if the room has lights it will turn them on if they are off and vice versa, if the lights are on the room should not be dark and if they are off it must be dark (Note not all rooms have lights and if they do not have lights they do require a flashlight to illuminate (for the look command to work)).

11) Give up: will exit the game and lose.

Introduction

The game implementation follows the clean code principles according to Robert Martin's book (Clean Code), the effective java principles according to Joshua Bloch's book (Effective Java), the SOLID programming principles, and satisfies Google styling guide.

Link for the code on github.

Robert martin's book "clean code"

In this section of the report I will cover how my code satisfies the clean code principles according Robert martin's book "clean code".

note: "chapter 5 of the book -formatting- was covered in the google style guide section- "

2.1) chapter 2: meaningful names.

1) Use meaningful, intention-revealing, pronounceable names. "page 17-18 in the book"

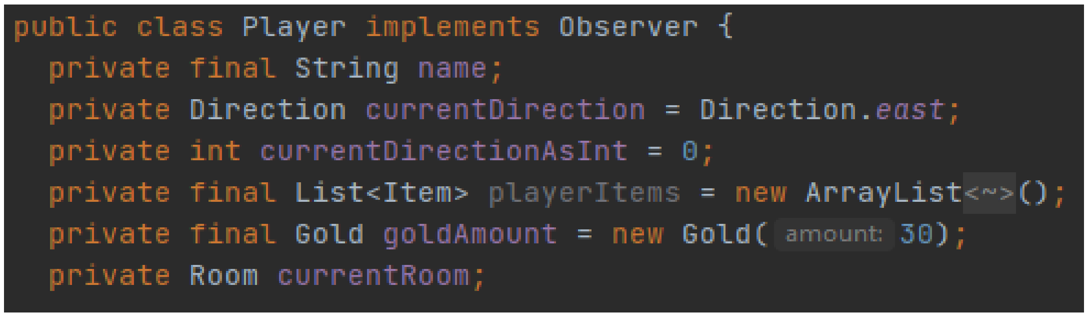
2) Use pronounceable names. "page 21 in the book"

3) Avoid disinformation and puns. "page 19-20 in the book"
   disinformation means: accidental similarities with something else entirely or too-subtle name differences.

   And puns mean: two different words for the same purpose.

4) use searchable names. "page 22 in the book"

   in this figure is an example of how my code satisfies the first four principles.



```
public class Player implements Observer {
  private final String name;
  private Direction currentDirection = Direction.east;
  private int currentDirectionAsInt = 0;
  private final List<Item> playerItems = new ArrayList<~>();
  private final Gold goldAmount = new Gold( amount: 30);
  private Room currentRoom;
```

Figure 1

Let us assume the first player object is firstPlayer.
-Then when trying to use the class instances from inside the class we write
firstplayer.name
firstplayer.currentDirection
-And from outside the class
firstplayer.getCurrentDirection()
firstplayer.getPlayerItems()

as you can see, the names I used are meaningful, intention-revealing, pronounceable and searchable names, and I also avoided disinformation and puns.

5) Don't be afraid to globally change bad names (including their uses, of course).
I refactored the names of the variables, methods, classes, and the packages to satisfy the clean code good naming practices.

6) try to avoid prefixes in names. "page 24 in the book"
for example, do not use names like X_position

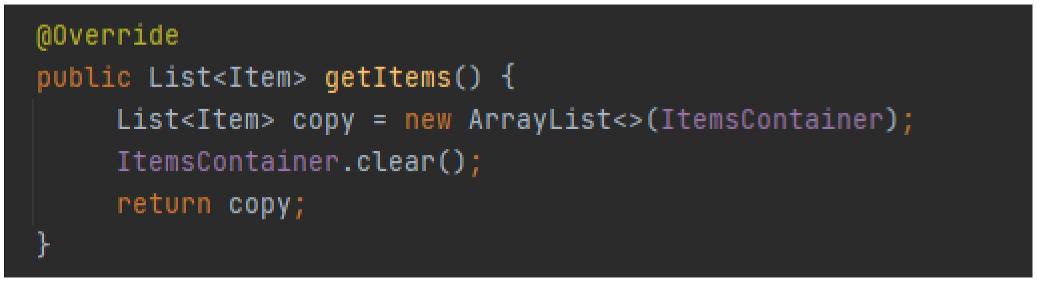
7) be clever with naming but not cute. "page 26 in the book"
In naming choose clarity over entertainment. Because cuteness in code and names often appear in the form of slang, which is what I did in choosing the names for my variables, methods, and classes.

2.2) chapter 3: functions.

1) Functions (in java language we use the term methods):
a) should be small (less than 100 lines).
b) do only one thing.

"page 34-36 in the book"

Here is an example show in the figure(2) from one of my methods that satisfies this point.

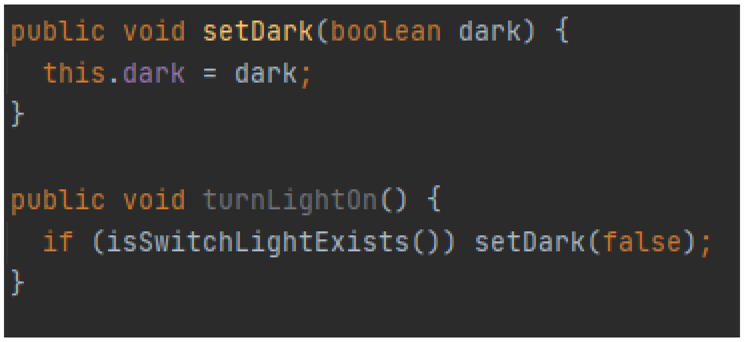


```
@Override
public List<Item> getItems() {
    List<Item> copy = new ArrayList<>(ItemsContainer);
    ItemsContainer.clear();
    return copy;
}
```

Figure 2

2) Order the functions thus broken down in depth-first order, so that the overall code can be read top-to-bottom. It is hard to overestimate the importance of descriptive and consistent names and of the absence of surprising side-effects.

"page 34-39 in the book"



```
public void setDark(boolean dark) {
  this.dark = dark;
}

public void turnLightOn() {
  if (isSwitchLightExists()) setDark(false);
}
```

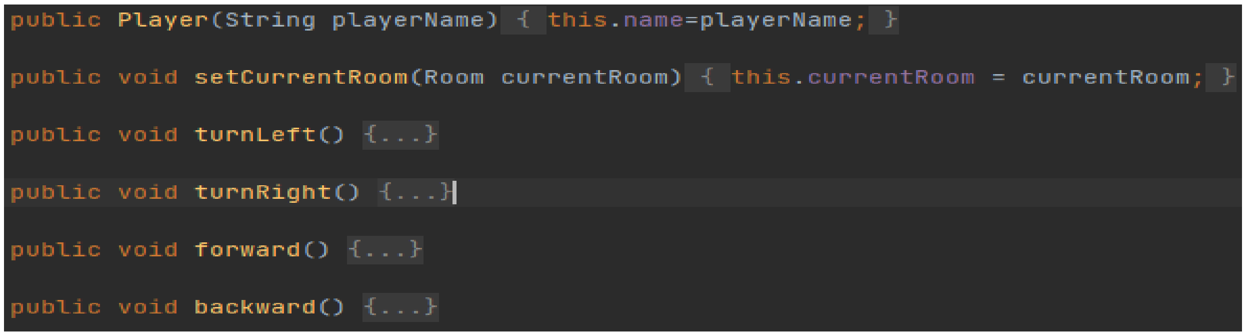
Figure 3

3) Parameters make functions harder to grasp and often are on a lower level of abstraction, so avoid them as best you can.

"page 40-42 in the book"

Unfortunately, we can't always avoid using parameters, but I made sure to use the least possible numbers of parameters.

The longest set of parameters in the methods in my code are only 1 parameter maximum.
Note: All this describes a good end result. Initially, you may well have long, ill-named, complex, parameter-rich functions that do many things. This is no problem, as long as you then go and refactor, refactor, refactor.



```
public Player(String playerName) { this.name=playerName; }

public void setCurrentRoom(Room currentRoom) { this.currentRoom = currentRoom; }

public void turnLeft() {...}

public void turnRight() {...}

public void forward() {...}

public void backward() {...}
```

Figure 4

2.3) Chapter 4: Comments.

1) The general idea is "The proper use of comments is to compensate for our failure to express ourselves in code." Comments do not make up for bad code, rather, we should express ourselves in the code”. “page 53-55 in the book”

2) Types of good comments are legal comments, informative comments, explanations of intent, warning of consequences, TODO, marking as important, documenting public API.

3) Types of bad comments are: unclear mumbling, redundant comments, misleading comments, mandated comments, changelog comments, a comment instead of putting code into a separate function, banners, closing-brace (etc.) comments, attributions 1 and by-lines, commented-out code, HTMLified comments, nonlocal information, too much or irrelevant information, comments needing explanation, documenting non-public API. “page 59-73 in the book”

Note: Chapter 5 is covered in the google style guide section.

2.4) Chapter 6: Objects and Data Structures.

1)Decide consciously what to hide in your objects. It depends on what changes are expected (and sometimes bare public data structures will be just fine).

2)Preferably, call only methods of your own class, of objects you have just created, of parameters, and of instance variables, not further methods reachable through these objects (Law of Demeter).

Consider three classes namely -- A, B, and C -- and objects of these classes -- objA, objB, and objC respectively. Now suppose objA is dependent on objB, which in turn composes objC. In this scenerio, objA can invoke methods and properties of objB but not objC.

The Law of Demeter principle takes advantage of encapsulation to achieve this isolation and reduce coupling amongst the components of your application. This helps in improving the code quality and promotes flexibility and easier code maintenance. The benefit of adhering to the

Law of Demeter is that you can build software that is easily maintainable and adaptable to future changes.

Consider a class C having a method M. Now suppose you have created an instance of the class C named O. The Law of Demeter specifies that the method M can invoke the following types of or a property of a class should invoke the following type of members only:

1) The same object, i.e., the object “O” itself

2) Objects that have been passed as an argument to the method “M”

3) Local objects, i.e., objects that have been created inside the method “M”

4) Global objects that are accessible by the object “O”

5) Direct component objects of the object “O”

2.5) Chapter 10: Classes.

1) Ordering: Constants before variables before methods (within each: public before private, but private methods used only once follow right after their usage).

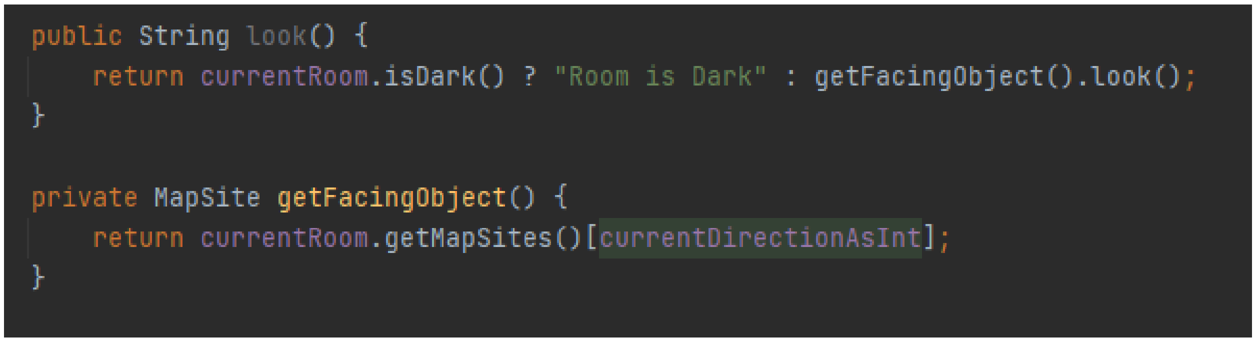

```java
public String look() {
    return currentRoom.isDark() ? "Room is Dark" : getFacingObject().look();
}

private MapSite getFacingObject() {
    return currentRoom.getMapSites()[currentDirectionAsInt];
}
```

Figure 5

2) Keep variables private unless that gets in the way of testing; then use protected or package.

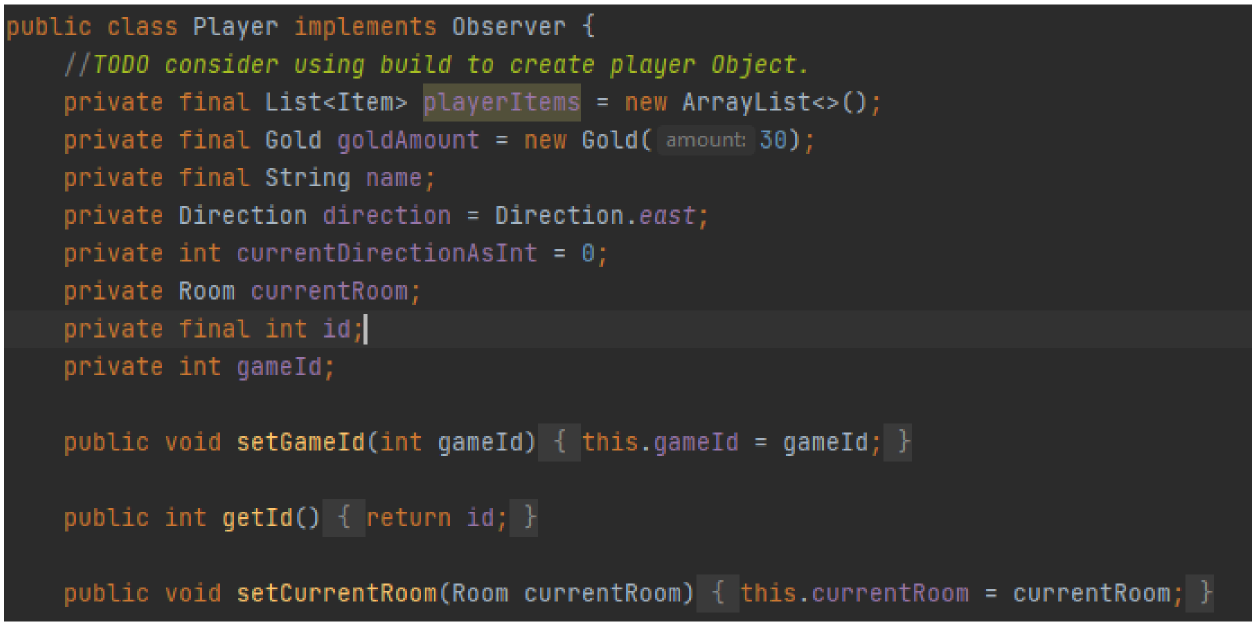
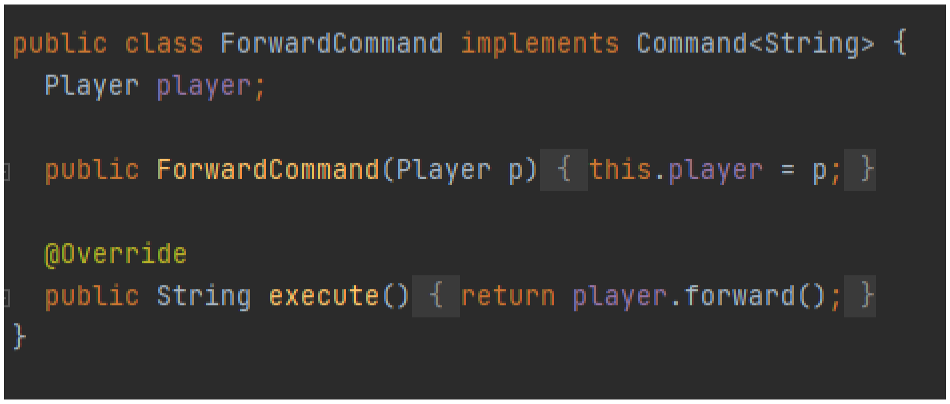

```java
public class Player implements Observer {
    //TODO consider using build to create player Object.
    private final List<Item> playerItems = new ArrayList<>();
    private final Gold goldAmount = new Gold( amount: 30);
    private final String name;
    private Direction direction = Direction.east;
    private int currentDirectionAsInt = 0;
    private Room currentRoom;
    private final int id;
    private int gameId;

    public void setGameId(int gameId) { this.gameId = gameId; }

    public int getId() { return id; }

    public void setCurrentRoom(Room currentRoom) { this.currentRoom = currentRoom; }
```

Figure 6

3) Classes should be small: Have only one responsibility (Single Responsibility Principle (SRP): have only one reason to change).

Dependency Inversion Principle (DIP): Rather than hard-coding calls to dependent services, rely on an abstraction (interface) only and pass a concrete service (object) in as a parameter.

Example from my code in figure 7



```java
public class ForwardCommand implements Command<String> {
  Player player;

  public ForwardCommand(Player p) { this.player = p; }

  @Override
  public String execute() { return player.forward(); }
}
```

Figure 7

2.8) Chapter 17: code smells and heuristics.

###) comments:

1)inappropriate information: meta-data such as authors, last-modified-date, PSR number, etc.... should not appear in comments. Comments must be reserved for technical noted about code and design.

2) obsolete comments: comments that has gotten old, irrelevant, or incorrect (they become misdirection in the code).

3)redundant comments: the comment that describe something clear or something that is explained by itself even if its correct (don't state the obvious).
Ex) i++ // increment i

4)commented-out code: who knows how old it is? Or if it has a meaning? Yet no one would dare to delete it assuming someone else needs it or has a plan for it.

###) functions (methods):

1)too many arguments: methods should have small number of arguments. No argument is best if possible.

No function in my code have more than 1 argument.

1) output arguments: readers expect the arguments to be input to the method **not output**. There are no output arguments in my code.

2) flag arguments: they loudly declare that the method has more than one thing to do, they are confusing and should not be used.

I never used flag arguments. Instead split my method into smaller methods each has one responsibility.

3) dead functions: functions that are never been called or used should be discarded.

All methods in my code have been used at least once.

###) general:

"pages 288-306"

1)multiple language in one source file:
I only used one language per file.

2)duplication: DRY principle (don't repeat yourself).
I have never done the same thing twice in my code. And I have never explained something clear or something that is explained by itself.

3)dead code: (code which is not executed) like if statement check a condition that can never happen.
there is no dead code in my solution.

4)use explanatory variables: to make the program readable, break the calculations up into more intermediate values that are held in variables with meaningful names.
example: your function takes seconds but for easy of reading you want the other people view it as minutes to make it more readable.
Int timeInseconds=timeInminutes*60;

5)avoid negative conditions.
No negative conditions in my code.

6)function should do one thing.
I avoided using the same function to do two things at a time if I needed put the second task in another method and call it from the first method

###) names:

Use unambiguous names.

For example when you use chest.isopen and it returns true you can interrupt exactly that the chest is open. So it is clear and unambiguous name.
another example when you call player.getDirection();its clear that it means that this method returns the direction of the player.

Jushua Bloch’s book “Effective Java”

Chapter 2: Creating and destroying objects.

1) **Avoid creating unnecessary objects.**

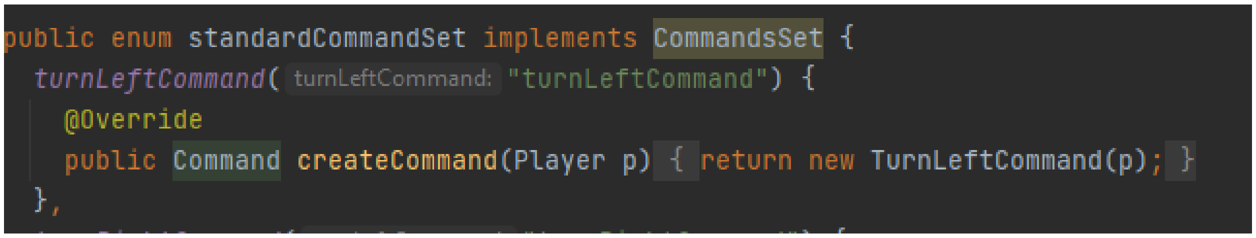
   It is often appropriate to reuse a single object instead of creating a new object every time.

   In my code, since class wall doesn’t have instances, I created only one wall to be set to the rooms sides that needs walls.

   Also, in the command pattern (for example let’s take the forwardCommand class) when the player move forward it doesn’t create another object of the class. Instead it uses the same object that was created.

2) Enforce the singleton property with a private constructor or an enum type.

This is an example of singilton that is responsible for creating commands object.



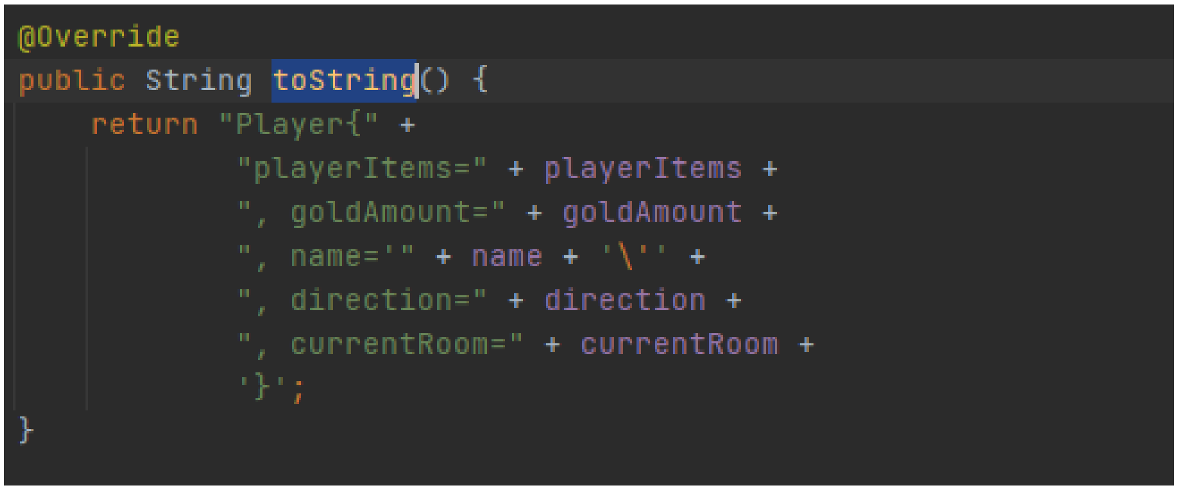
```
public enum standardCommandSet implements CommandsSet {
  turnLeftCommand( turnLeftCommand: "turnLeftCommand") {
    @Override
    public Command createCommand(Player p) { return new TurnLeftCommand(p); }
  },
```

Figure 8

Chapter 3: Method common to all objects.

1) Consider overriding *toString().*

The general contract for toString() says that the returned string should be “a concise but informative representation that is easy for a person to read". Providing a good toString implementation makes your class much more pleasant to use and makes systems using the class easier to debug.

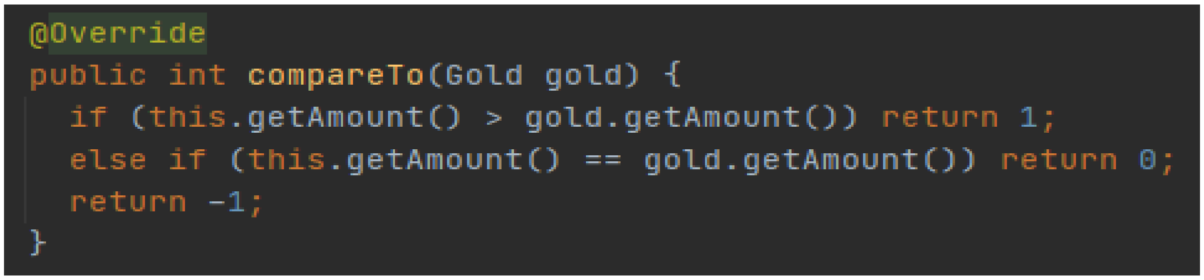


```java
@Override
public String toString() {
    return "Player{" +
            "playerItems=" + playerItems +
            ", goldAmount=" + goldAmount +
            ", name='" + name + '\'' +
            ", direction=" + direction +
            ", currentRoom=" + currentRoom +
            '}';
}
```

Figure 9

2) Consider overriding *compareTo().*

This is an example from my code overriding CompareTo() in figure 10.

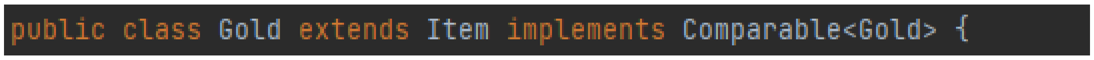


```java
@Override
public int compareTo(Gold gold) {
  if (this.getAmount() > gold.getAmount()) return 1;
  else if (this.getAmount() == gold.getAmount()) return 0;
  return -1;
}
```

Figure 10

3) Consider implementing Comparable.

Whenever you implement a value class that has a sensible ordering, you should have that class implement the comparable interface so that its instances can be easily sorted, searched and used in comparison-based collections.



```java
public class Gold extends Item implements Comparable<Gold> {
```

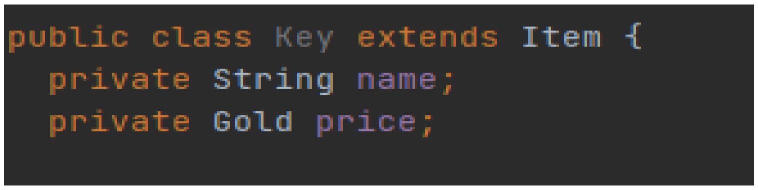
Figure 11

Chapter 4: Classes and interfaces.

a) Minimize the accessibility of classes and members.

A well-designed component hides all its implementation details, cleanly separating its API from its implementation. Components then communicate only through their APIs and are oblivious to each other's inner workings. This concept, known as information hiding or encapsulation, is a fundamental tenet of software design.

The rule of thumb is simple: make each class or member as inaccessible as possible. In other words, use the lowest possible access level consistent with the proper functioning of the software that you are writing.

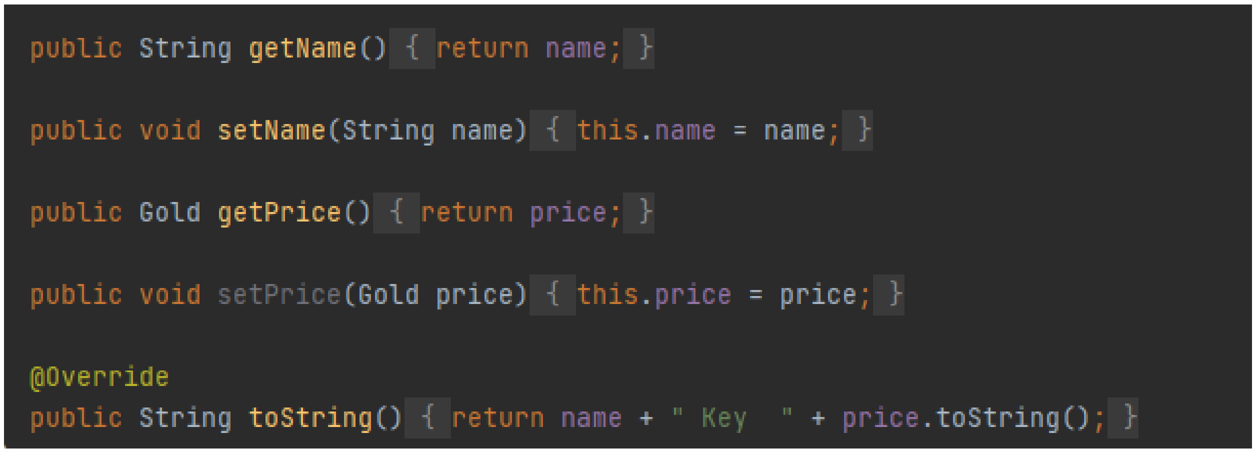


```
public class Key extends Item {
  private String name;
  private Gold price;
```

Figure 12

b) In public classes, use accessor methods, not public fields.

Public classes should never expose mutable fields. Its less harmful though still questionable, for public classes to expose immutable fields. furthermore

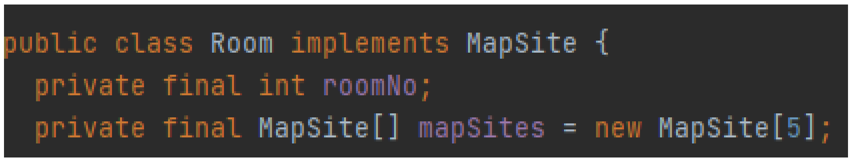


```
public String getName() { return name; }

public void setName(String name) { this.name = name; }

public Gold getPrice() { return price; }

public void setPrice(Gold price) { this.price = price; }

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

Figure 13

c) Favor composition over inheritance.

Inheritance in this case is when a class extends another (*implementation inheritance*) Not interface inheritance. (**Inheritance violates encapsulation).**



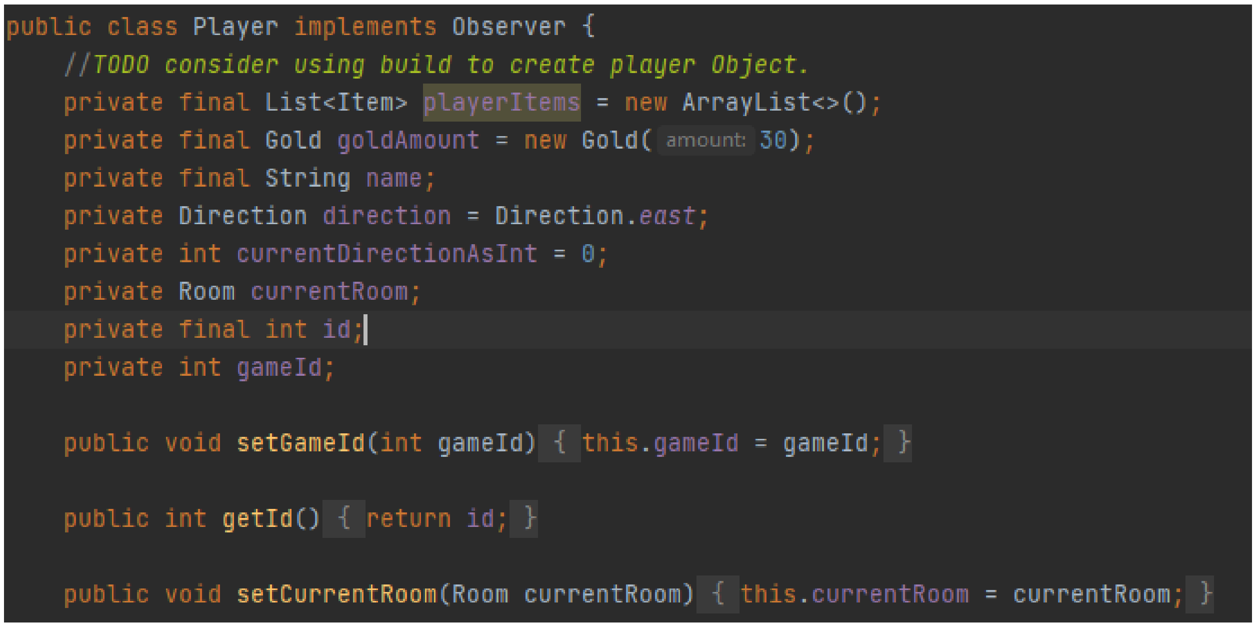
```
public class Room implements MapSite {
  private final int roomNo;
  private final MapSite[] mapSites = new MapSite[5];
```

Figure 14

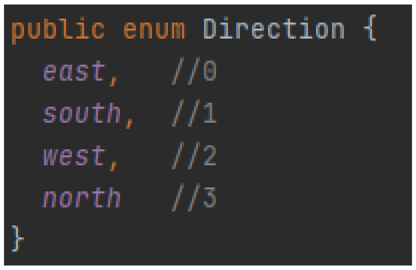
Since every room needs a floor, I make it as composition as the fifth element of the four sides not as inheritance.

d) Prefer interfaces to abstract classes:

Interfaces is generally the best way to define a type that permits multiple implementations.



```java
public class Player implements Observer {
    //TODO consider using build to create player Object.
    private final List<Item> playerItems = new ArrayList<>();
    private final Gold goldAmount = new Gold( amount: 30);
    private final String name;
    private Direction direction = Direction.east;
    private int currentDirectionAsInt = 0;
    private Room currentRoom;
    private final int id;
    private int gameId;

    public void setGameId(int gameId) { this.gameId = gameId; }

    public int getId() { return id; }

    public void setCurrentRoom(Room currentRoom) { this.currentRoom = currentRoom; }
```

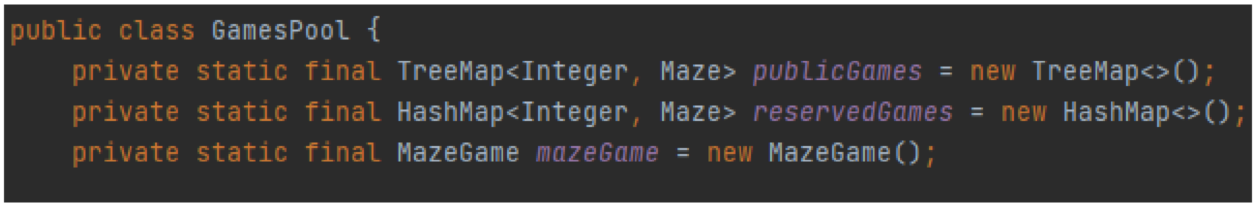


```java
public enum Direction {
  east,   //0
  south,  //1
  west,   //2
  north   //3
}
```

at first I used direction to return the direction as a string and used curruntDirectionAsInt to return the current direction as int value but decided to make direction enum and instead of using the variable curruntDirectionAsInt (to be used as an index for the mapSite or RoomSide arrays) and implement a function to convert between them I can use the ordinals if I use enum. So curruntDirectionAsInt and the functions to convert the values was discarded from the code. The code becomes smaller and more readable

e) Favor static member classes over non-static.

If a member class does not need access to its enclosing instance, then declare it static. If the class is non static, each instance will have a reference to its enclosing instance. That can result in the enclosing instance not being garbage collected and memory leaks.


```
public class GamesPool {
    private static final TreeMap<Integer, Maze> publicGames = new TreeMap<>();
    private static final HashMap<Integer, Maze> reservedGames = new HashMap<>();
    private static final MazeGame mazeGame = new MazeGame();
```

Chapter 5: Generics

1) Don't use raw types in code:
   A raw type is a name for a generic interface or class without its type argument.

   `List list = new ArrayList(); // raw type`
   instead use parameterized types.

2) Prefer lists to arrays
   Arrays are reified: Arrays know and enforce their element types at runtime. Generics are erasure: Enforce their type constrains only at compile time and discard (or erase) their element type information at runtime. It's always prefer to catch the error sooner than later so prefer Lists.
   So I only used array carefully and only when the size is fixed.

3) Eliminate unchecked warnings

   Eliminate every unchecked warning that you can. If you can't eliminate it, but you can prove that the code that provoked the warning is type safe, then suppress the warning with an @SuppressWarnings("unchecked") annotation and also add a comment why it's safe to do. Always use the SuppressWarnings on the smallest scope as possible.

   I have eliminated most of them and commented why the others are safe.

Chapter 7: Lambdas and streams

1) **Prefer Lambdas to anonymous classes.**

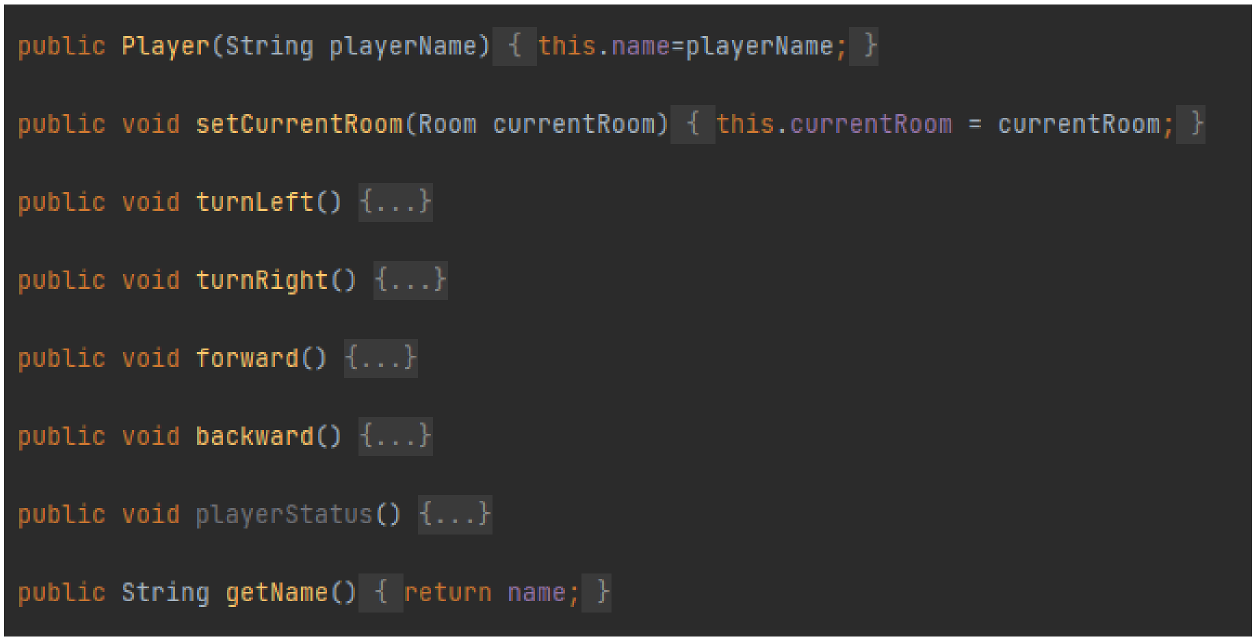
   Lambdas are the best way to represent function objects. As a rule of thumb, lambdas need to be short to be readable. Three lines seems to be a reasonable limit. Also, the lambdas need to be self-explanatory since it lacks name or documentation. Always think in terms of readability.

2) **Prefer methods references to lambdas**

   Method references often provide a more succinct alternative to lambdas. Where method references are shorter and clearer, use them; where they are not, stick with lambdas.

Chapter 8: Methods

A) Design method signatures and name methods carefully
   also avoid long parameter list.
   I choose the most intention-revealing names for my methods ands avoid the long parameters lists. The longest parameters set in any method is one at maximum.
   This is an example of how my code satisfies this principle:



```
public Player(String playerName) { this.name=playerName; }

public void setCurrentRoom(Room currentRoom) { this.currentRoom = currentRoom; }

public void turnLeft() {...}

public void turnRight() {...}

public void forward() {...}

public void backward() {...}

public void playerStatus() {...}

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

B) Return empty arrays or collections, not nulls.
Never return null in place of an empty array or collection. It makes your API more difficult to use more prone to error, it has no performance advantages.

Google style Guide

1)formatting.

a) braces:

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace.

b) Block indentation: +2 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block.

This comes by default from google styling tool.

c) One statement per line

d) Column limit: 100 Characters

All lines are less than 50 characters in length in my code

e) *Horizontal whitespace*

1) Separating any reserved word from an open parenthesis “ (” or a closing curly brace “ } “.

2) Between the type and variable of a declaration: `List<String>   list`

f) One variable per declaration

Every variable declared in a special line in my code

g) Class names are written in Upper-Camel-Case.
Example from class Mirror.

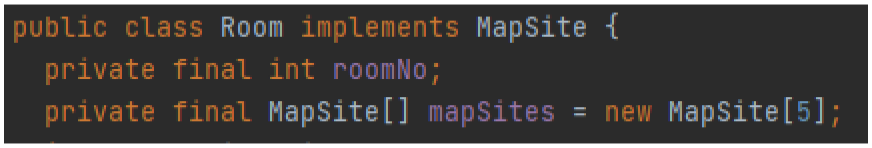
h) Method names, Parameter names and variable names must be written in lower-Camel-Case.

Data structures

1) Arrays.
   I used arrays in the class “Room” instead of Arraylists for the following reasons.

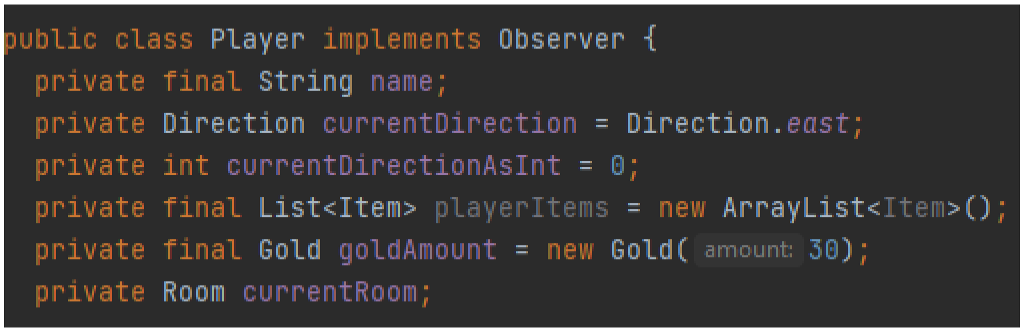
   a) ArrayLists usually allocate about twice the memory you need now in advance so that you can append items very fast. So, if you use ArrayLists there will be wastage of memory if you are never going to add any more items (since we know that the number of room sides is fixed, I prefer using arrays).

   b) I prefer the more straightforward [ ] syntax for accessing elements over ArrayList's get(), set(), add() and remove(). This really becomes more important when I need multidimensional arrays (like how I used it in project 1 to implement the map).



```java
public class Room implements MapSite {
  private final int roomNo;
  private final MapSite[] mapSites = new MapSite[5];
```

2) ArrayLists.



```java
public class Player implements Observer {
  private final String name;
  private Direction currentDirection = Direction.east;
  private int currentDirectionAsInt = 0;
  private final List<Item> playerItems = new ArrayList<Item>();
  private final Gold goldAmount = new Gold( amount: 30);
  private Room currentRoom;
```

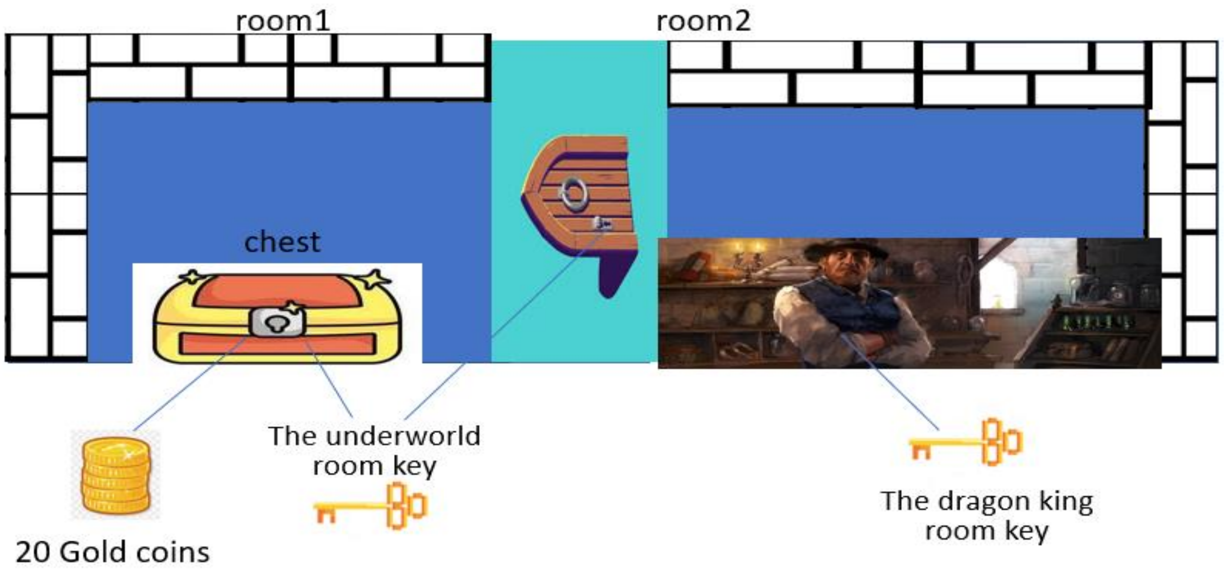
I also used the same implementation of ArrayList in classes “seller” and “chest”.

I used ArrayLists in the code in these places for the following two reasons:
Lists can easily grow in size (better to use it that arrays since the size you need is not fixed -dynamic size-), and you can add and remove items in the middle of the list easily. That cannot be done easily with arrays. (when a player sells an item then you need to remove it from the player list and add the item to the list of items of the buyer).

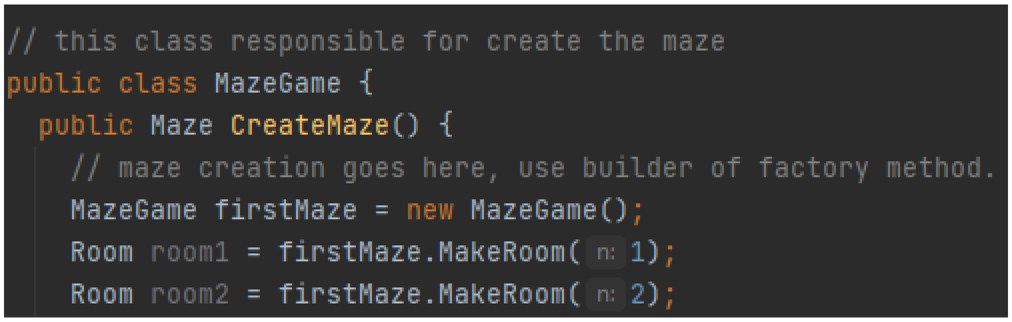
How to create a maze

The “MazeGame” class is the class which is responsibility is creating the maze.

Let’s create the room in the fig bellow together.



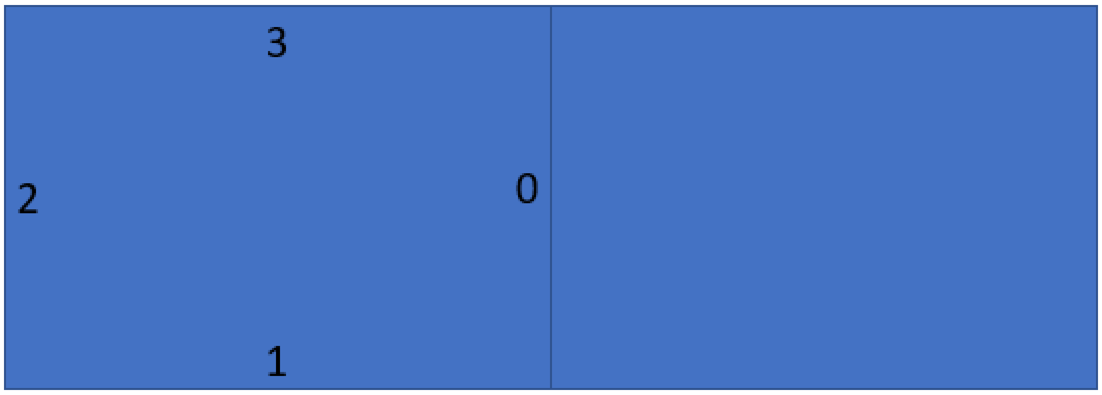
1) **The First thing to do is to create an object of type "MazeGame" then use the factory method to create objects of type room.**



```
// this class responsible for create the maze
public class MazeGame {
  public Maze CreateMaze() {
    // maze creation goes here, use builder of factory method.
    MazeGame firstMaze = new MazeGame();
    Room room1 = firstMaze.MakeRoom( n: 1);
    Room room2 = firstMaze.MakeRoom( n: 2);
```

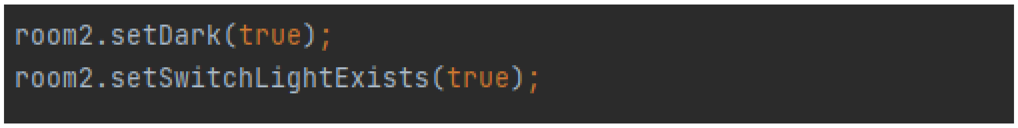
Keep in mind that the direction ordinal is what I used to set the sides

e.g. the east direction ordinal is zero. the south direction ordinal is one.
the west direction ordinal is two. the north direction ordinal is three.



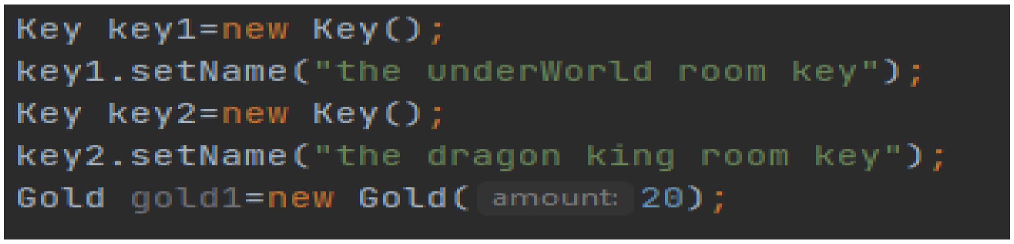
room1 room2

2) **set the rooms properties (dark rooms and if they have switch lights).**

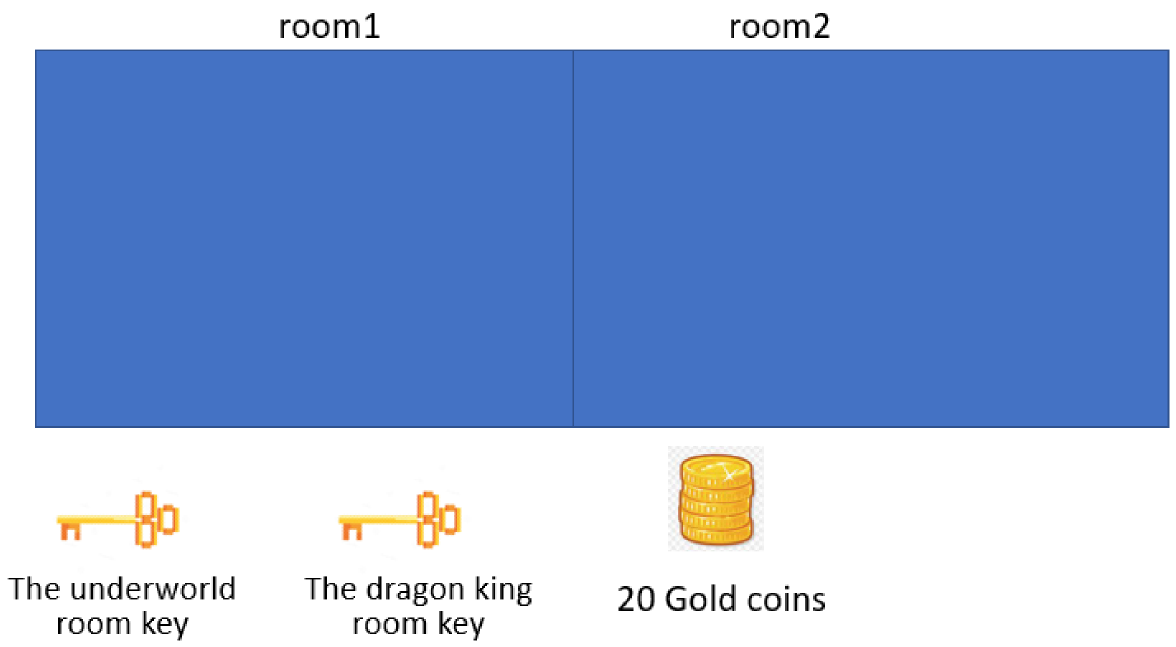


```
room2.setDark(true);
room2.setSwitchLightExists(true);
```

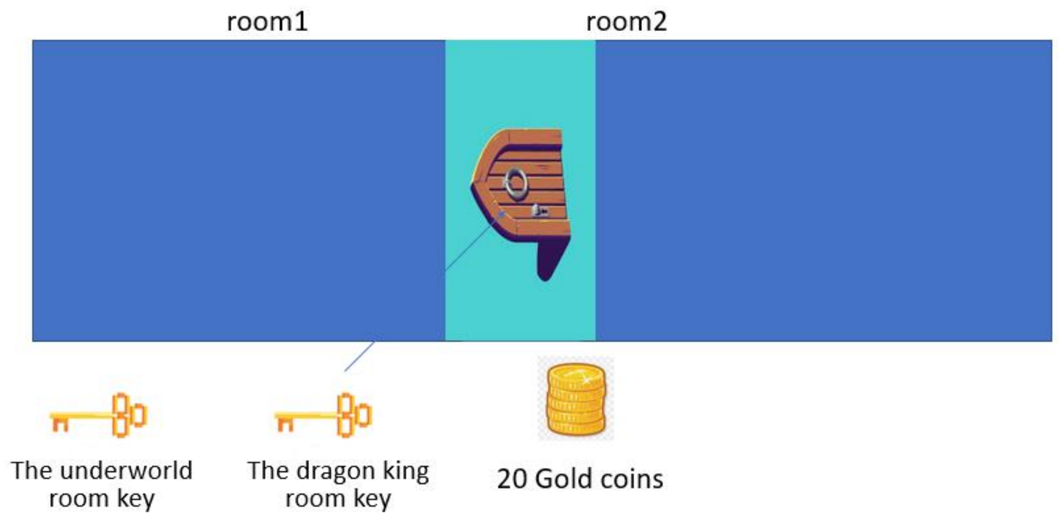
3) **create the items you want the room sides to have.**
for example you need to create keys you want a seller or a painting or a chest to have in the room.



```
Key key1=new Key();
key1.setName("the underWorld room key");
Key key2=new Key();
key2.setName("the dragon king room key");
Gold gold1=new Gold( amount: 20);
```



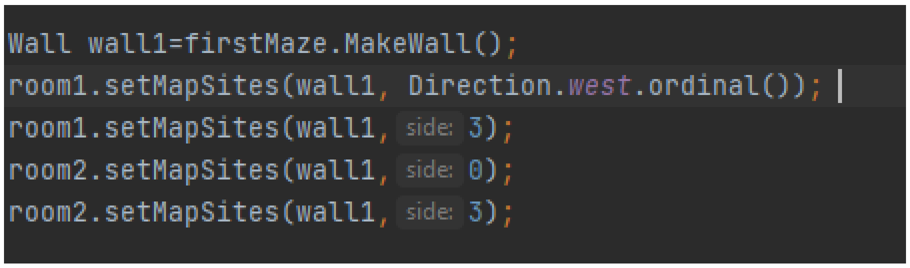
4) **create a door (a class responsible to link between the rooms) using the factory method and give it the two rooms you want to link together through this door.**
also, you can set the door properties if you want it closed, locked , and you want a specific key to unlock it.



The two rooms have the same door (reference to the same door) which is set to one of the sides of each room (you can assume it is just like a link between the two rooms).

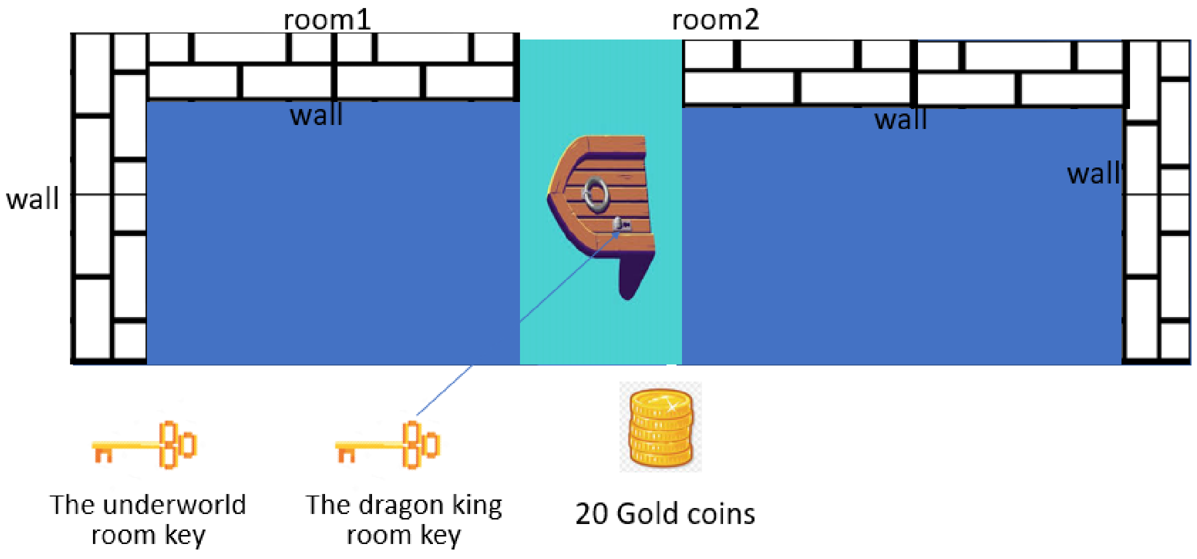
5) **one wall and link it to the sides of the room you want to have.**
Note: yes one wall to all the sides you want to have it and does not matter if there are more rooms since all the walls are the same.

In “effective java” book, it says “Avoid creating unnecessary objects”

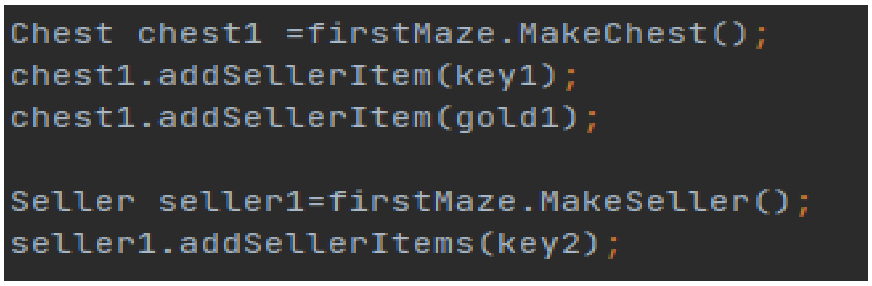


```
Wall wall1=firstMaze.MakeWall();
room1.setMapSites(wall1, Direction.west.ordinal());
room1.setMapSites(wall1, side: 3);
room2.setMapSites(wall1, side: 0);
room2.setMapSites(wall1, side: 3);
```

Remember that we are using the ordinals of the enum “direction”.

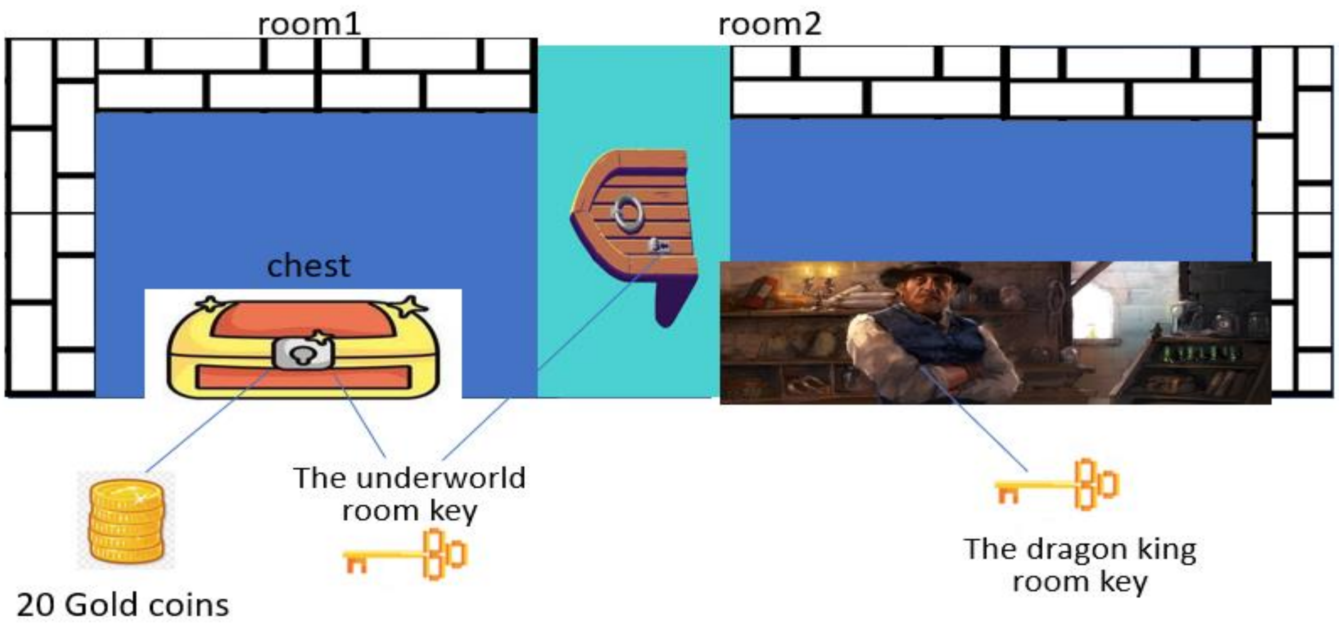


6) **create the other sides of the room using the factory method and to link each item to the specific side you want it to have the item.**

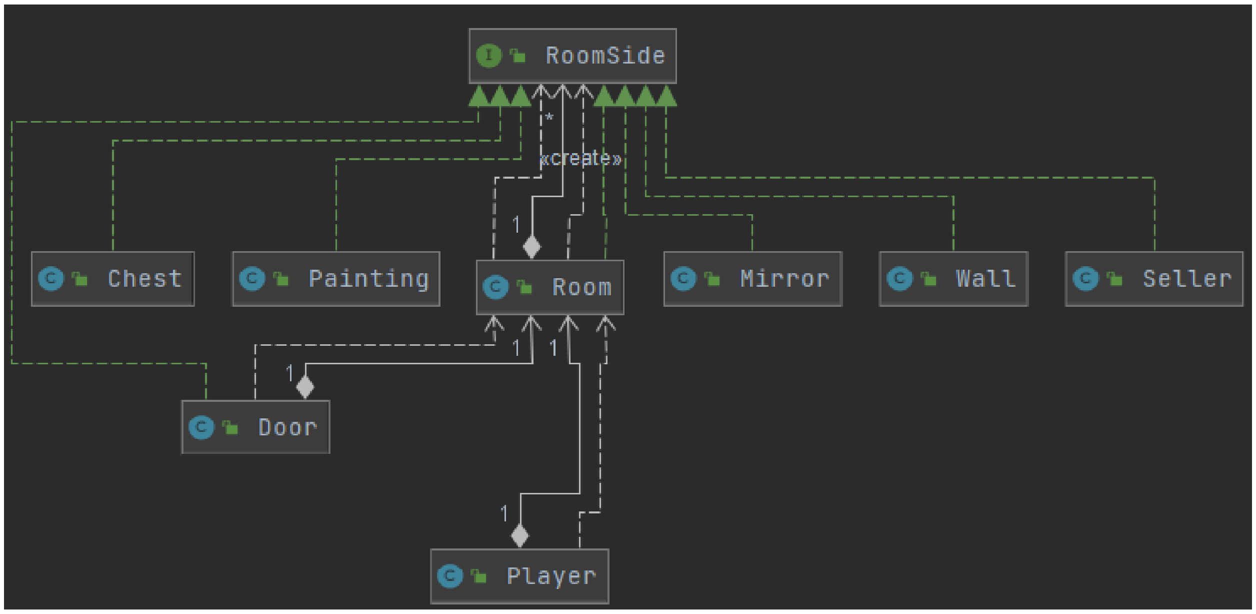


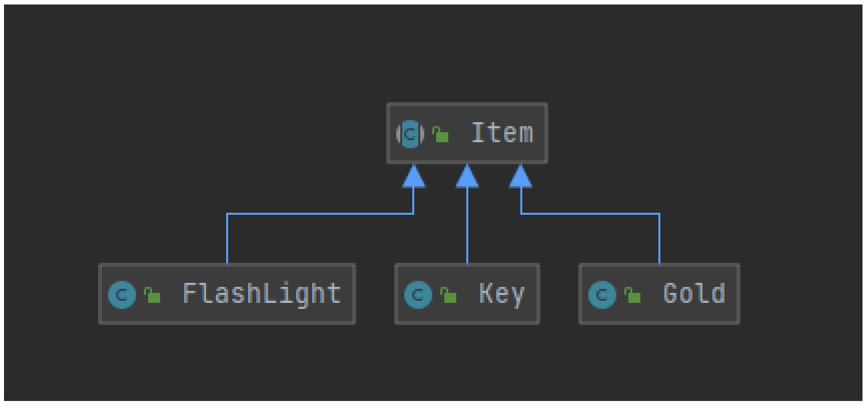
```
Chest chest1 =firstMaze.MakeChest();
chest1.addSellerItem(key1);
chest1.addSellerItem(gold1);

Seller seller1=firstMaze.MakeSeller();
seller1.addSellerItems(key2);
```

And as a result to the above code we create the rooms we desired.


room1
room2
chest
20 Gold coins
The underworld room key
The dragon king room key

UML


RoomSide
Chest
Painting
Room
Mirror
Wall
Seller
Door
Player
«create»
*
1
1
1
1
1

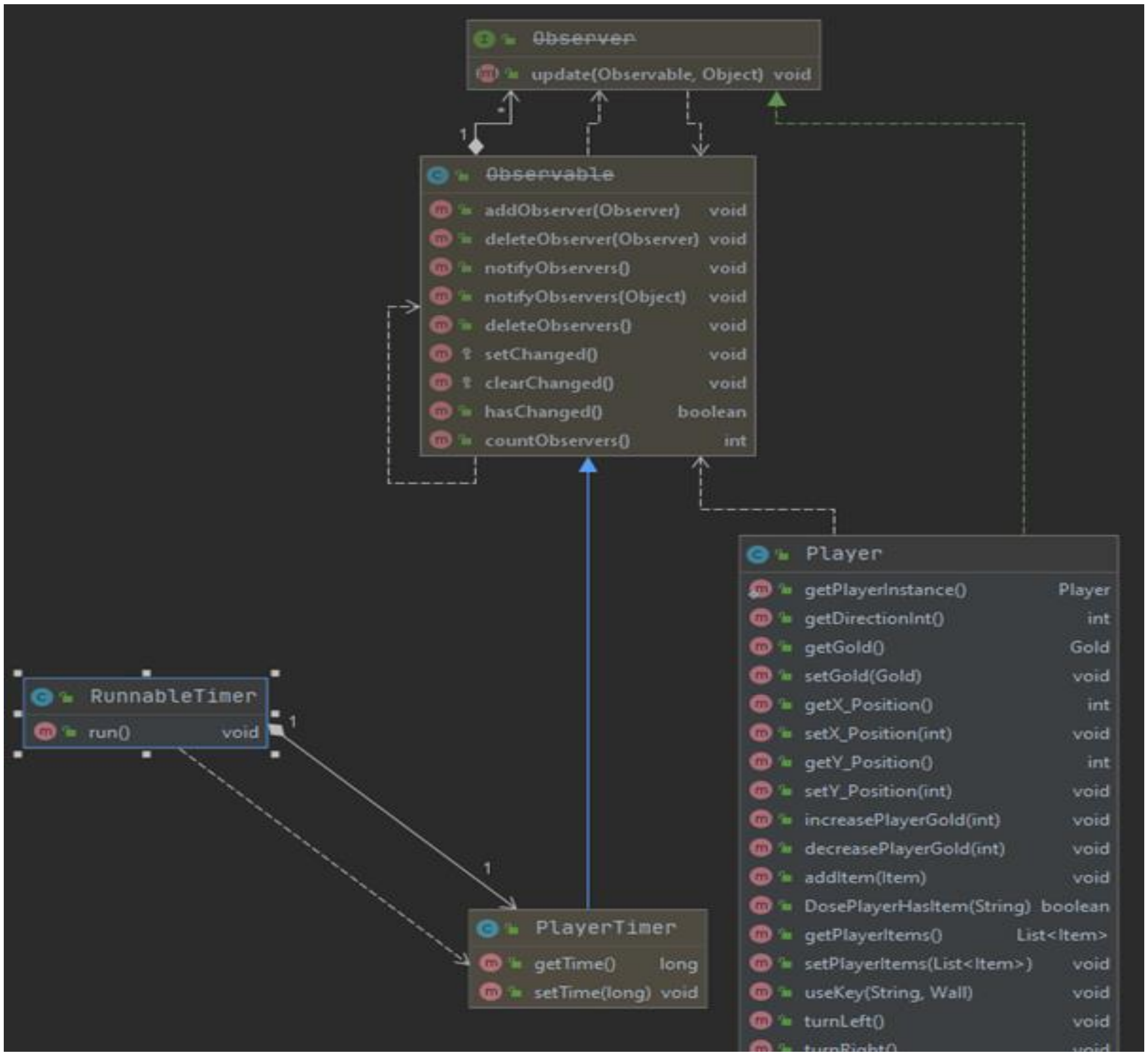

Item
FlashLight
Key
Gold

Design patterns

1) Thread safe Observer design pattern:

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.



2) Command pattern:

Command pattern is a data driven design pattern as and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

3) Singleton pattern:

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

create a *Single-Object* class. *Single-Object* class have its constructor as private and have a static instance of itself.

Single-Object class provides a static method to get its static instance to outside world. our class will use *Single-Object* class to get a *Single-Object* object.

Factory method Design pattern

State pattern Design pattern

SOLED principles

Single-Responsibility Principle

Single-responsibility Principle (SRP) states:

A class should have one and only one reason to change, meaning that a class should have only one job.

In my code, each class is responsible about one thing only as you will see in the examples below.

Class timer: is responsible to end the game if timer reaches zero.

Class player: keeps the player information up to date.