Name: Issa Anagreh

Project: World Navigator

◆ How to play:

- When the app starts, it asks about which map would you like to play by choosing the number of it -originally 1 map is available-.

- Each room has a size of 5x5 squares, use <room> command to have fun with real room simulator.

	a	b	c	d	e
1	a1	b1	c1	d1	e1
2	a2	b2	c2	d2	e2
3	a3	b3	c3	d3	e3
4	a4	b4	c4	d4	e4
5	a5	b5	c5	d5	e5

figure-1: This figure can be assumed as a chess board

- The player starts in a certain square in a certain room then starts to walk forward or backward with rotating himself left and right.

- To tell the available commands, the player can use <commands>.

- <open> command is the only command that can change the room from one to another in front of an open door.

- <check> can check what is the item you are in front of or can acquire checkable items' contents.

- Forwarding and back warding, with left and right, is boring? use <setloc> cheating command with the required square tag (ex: c2) to change player location directly.

- <look> is a great command to tell what items the opposite wall has with their locations.

- For locked items should use <useKey> before checking them or opening

them.

- Some checkable items (ex: chests) can be locked but has nothing inside.
- Some openable items (ex: doors) can be opened to nothing.

◆ Map .json:

- Each map needs its own .json file.
- “name”, “file”, “end_time” **must** be added.
- Player details are optional.
- Rooms **must** at least have 1 room.
- Rooms can be named by default.
- Rooms are lit by default.
- Rooms do not have a light switch by default (false value).
- 4 walls are **must** be added to each room with the names of (n_wall, e_wall, s_wall, and w_wall).
- Walls can have nothing.
- Wall items' names are **not changeable** such as (door, chest, mirror, painting, and seller).
- Items’ name is **not optional**.
- Items’ key is **optional** but can only be added to doors and chests, which refers to an item’s opening key.
- Items’ location is a **must** with a format of “letter between a to e + number between 1 to 5”, and double-check the location with figure-1.
- Items’ content map is **optional**.
- The contents can have keys, flashlights, or golds all optionally but with **strictly** named (keys, flashLights, golds).
- The contents can not be meaningful if added to the doors.
- Sellers can have content that can be bought by the player, and a selling map of content can be sold by the player with their prices such as (“keys”: 7) which means a player can sell a key by 7 golds, but with strictly named of (keys, flashLights).

- Use tests to check if a new made map is correctly done.

◆ Rooms:

- Dimensions:

- Each room has a size of 5x5 squares, the player starts in a certain square then starts to walk forward or backward with rotating himself left and right.

	a	b	c	d	e
1	a1	b1	c1	d1	e1
2	a2	b2	c2	d2	e2
3	a3	b3	c3	d3	e3
4	a4	b4	c4	d4	e4
5	a5	b5	c5	d5	e5

figure-1: This figure can be assumed as a chess board

- Rooms' items:

- Adding items to a room needs a double check of the location of this item depending on figure-1.

- Items **cannot** be overlapped in the same location.

- Items must be laid to a wall.

- Use tests to check if a new room is correctly done.

◆ Items:

- Openable items:

- New openable item (ex: a gate) can be added smoothly, by theses steps:

- let us pretend we have this item details:

```
"gate": {
  "name": "room_5-gate_1",
  "is_locked": "false",
  "to": "room_2",
  "golden": "false",
  "location": "e5"
}
```

- Adding a preparing method of few new lines in ItemsFactory:

```
private void prepareGate(JSONObject wall) {
  Object gate = wall.get("gate");
  if (gate != null) {
    items.put("gate", new Gate(castToJSONObject(gate)));
  }
}
```

- Obviously now we will need a Gate class extends Item and implements NextGoing (report has more info):

```java
public class Gate extends Item implements NextGoing {
    public final String NAME;
    private final Boolean IS_GOLDEN;
    private final String NEXT_ROOM;
    private final String LOCATION;

    public Gate(JSONObject gate) {
        this.NAME = (String) gate.get("name");
        this.LOCATION = (String) gate.get("location");
        this.IS_GOLDEN = gate.get("golden").equals("true");
        this.NEXT_ROOM = (String) gate.get("to");

        if (gate.get("key") != null) {
            super.setUseKeyBehavior(new Openable(gate, name: "Gate"));
        }
        super.setCheckBehavior(new Uncheckable());
    }

    @Override
    public String getNextRoom() {
        if (super.useKeyBehavior != null && super.useKeyBehavior.getIs_lock
            return "locked";
        } else {
            if (this.getGolden() != null && this.getGolden()) {
                return "golden";
```

Do not forget to implement all of the abstract methods. THAT'S IT.

- Uncheckable items:

- New uncheckable item (ex: a window) can be added smoothly, by theses steps:

- let us pretend we have this item details:

```
"window": {
  "location": "d1"
}
```

- Adding a preparing method of few new lines in ItemsFactory:

```
private void prepareWindow(JSONObject wall) {
  Object window = wall.get("window");
  if (window != null) {
    items.put("window", new Window(castToJSONObject(window)));
  }
}
```

- Obviously now we will need a Window class extends Item:

```
public class Window extends Item {
  public final String NAME;
  private final String LOCATION;

  public Window(JSONObject window) {
    this.NAME = "Window";
    this.LOCATION = window.get("location").toString();

    super.setCheckBehavior(new Uncheckable());
  }

  @Override
  public String getLocation() { return this.LOCATION; }

  @Override
  public String getName() { return this.NAME + " Looks to the blue sky"; }

  @Override
  public String getType() { return "Window"; }

  @Override
  public int compareTo(String location) { return this.getLocation().compareTo(location); }
```

Do not forget to implement all of the abstract methods. THAT'S IT.

- Unlocked_Checkable items:

- New unlocked_chackable item (ex: a table) can be added smoothly, by theses steps:

- let us pretend we have this item details:

```
"table": {
  "name": "room_2-table_1",
  "is_locked": "false",
  "content": {
    "golds": 1
  },
  "location": "d1"
}
```

- Adding a preparing method of few new lines in ItemsFactory:

```
private void prepareTable(JSONObject wall) {
  Object table = wall.get("table");
  if (table != null) {
    items.put("table", new Table(castToJSONObject(table)));
  }
}
```

- Obviously now we will need a Table class extends Item:

```
public class Table extends Item {
    private final String LOCATION;
    private final String NAME = "Table";

    public Table(JSONObject table) {
        this.LOCATION = (String) table.get("location");
        if (table.get("existed").equals("true")) {
            super.setCheckBehavior(new Unlocked_Checkable(table, this.LOCATION)
        }
    }

    public String getLocation() {
        return this.LOCATION;
    }

    @Override
    public String getName() {
        return this.NAME;
    }

    public String getType() { return "table"; }

    @Override
    public String toString() { return "Table: " + this.NAME + " in: " + this.LO
```

Do not forget to implement all of the abstract methods. THAT'S IT.

- Locked_Checkable items:

- Adding new openable item (ex: a safe) can be added smoothly, by theses steps:

- let us pretend we have this item details:

```
"safe": {
  "name": "room_1-safe_1",
  "is_locked": "true",
  "key": "key-room_1-safe_1",
  "content": {
    "golds": 10
  },
  "location": "c5"
}
```

- Adding few new lines in ItemsFactory constructor:

```
Object safe = wall.get("safe");
if (safe != null && (((JSONObject) safe).get("existed")).equals("true")) {
  items.put("safe", new Safe((JSONObject) safe));
}
```

- Obviously now we will need a Safe class extends Item:

```java
public class Safe extends Item {
  private final String NAME;
  private final String LOCATION;

  public Safe(JSONObject safe) {
    this.NAME = (String) safe.get("name");
    this.LOCATION = (String) safe.get("location");

    if (safe.get("key") != null) {
      super.setUseKeyBehavior(new Openable(safe,  name: "Safe"));
      if (safe.get("existed").equals("true")) {
        super.setCheckBehavior(new Locked_Checkable(safe, this.LOCATION, super.useKeyBel
      }
    } else {
      if (safe.get("existed").equals("true")) {
        super.setCheckBehavior(new Unlocked_Checkable(safe, this.LOCATION));
      }
    }
  }
```

Do not forget to implement all of the abstract methods. THAT'S IT.

- **P.S:** Optional special commands can be added to the PlayerController if new special Items are added.

◆ Content:

- Already existed contents are such as (golds, flash lights, keys).

- Let's pretend wanting to add a new content and call it a **masterKey** that can open any closed door but only once.

- Starting with the adding this content to the room_1-chest_2.

```
"w_wall": {
  "chest": {
    "name": "room_1-chest_2",
    "is_locked": "false",
    "key": "",
    "content": {
      "keys": [
        "key-room_1-door_1",
        "key-room_1-chest_1",
        "key-room_1-safe_1"
      ],
      "flashLights": 0,
      "golds": 5,
      "masterKeys": 1
    },
    "location": "a4"
  }
}
```


- Obviously now we will need a MasterKey class extends Item implements KeyChecker (unlock method is always true):

```java
public class MasterKey extends Key implements KeyChecker {
  private final String NAME;

  public MasterKey() {
    super( name: "master");
    this.NAME = "master";
  }

  @Override
  public String getName() { return this.NAME; }

  @Override
  public Boolean unlock(Openable openable) { return true; }

  @Override
  public String toString() { return this.getName(); }
}
```

- A use_mastherKey method is needed in PlayerModel:

```java
public void use_masterKey() {
  String print = "";
  List<KeyChecker> masterKeysList = new ArrayList<>();
  String masterKeysString = "masterKeys";
  if ((castToInt(this.contents.get(masterKeysString))) > 0) {
    masterKeysList.add(new MasterKey());
    Item item = this.wall.itemsFactory.getItem(this.location);
    print =
        item.toString().equals("Space") ? "Opening nothing" : item.applyUseKey(masterKeysList);
    this.contents.put(masterKeysString, castToInt(this.contents.get(masterKeysString)) - 1);
  } else {
    print = "You have no master keys";
  }
  notify_player(print);
}
```

- Let’s assign a master key command to playerController:

```
public PlayerController(PlayerModel playerModel) {
  super.playerModel = playerModel;

  super.commandsMap.put("o", PlayerControllerInterface::myOrientation);
  super.commandsMap.put("loc", PlayerControllerInterface::myLocation);
  super.commandsMap.put("l", PlayerControllerInterface::rotateLeft);
  super.commandsMap.put("r", PlayerControllerInterface::rotateRight);
  super.commandsMap.put("f", playerController -> move(PlayerControllerMaster.MoveParam.forward));
  super.commandsMap.put("b", playerController -> move(PlayerControllerMaster.MoveParam.backward));
  super.commandsMap.put(
      "c",
      playerController -> {
        this.check();
        this.acquire_items();
      });
  super.commandsMap.put("items", PlayerControllerInterface::myItems);
  super.commandsMap.put("key", PlayerControllerInterface::use_key);
  super.commandsMap.put("light", PlayerControllerInterface::switchLights);
  super.commandsMap.put("flash", PlayerControllerInterface::flashLight);
  super.commandsMap.put("master", playerController -> super.playerModel.use_masterKey());
  super.commandsMap.put("setloc", PlayerControllerInterface::setLocation);
}
```