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.

	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 open 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’ existed is a must with values of (true, false).

- 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 content map can have keys, flashlights, or golds all optionally but with strictly named (keys, flashLights, golds).

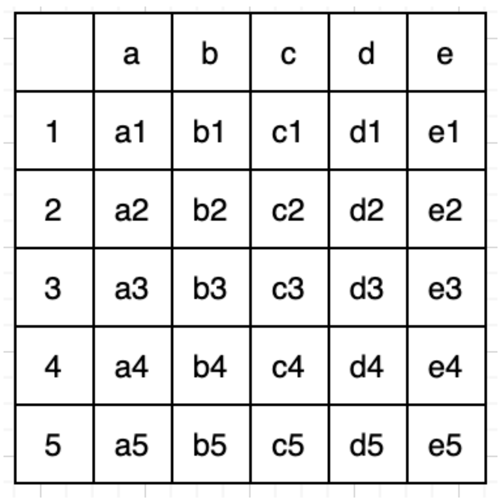
- The content map can not be meaningful if added to the doors.

- Sellers can have content that can be bought by the player, and a selling map of items 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).

- **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.

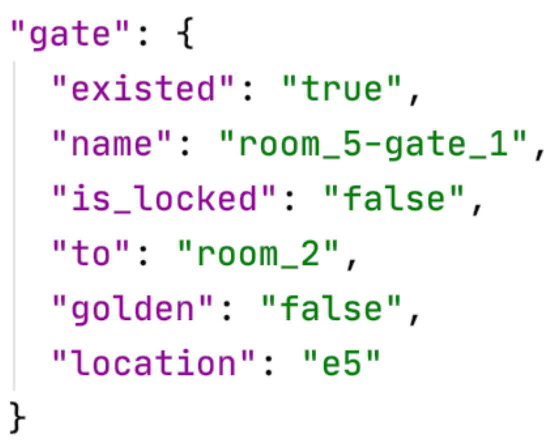
- Items **cannot** be overlapped in the same location.

- **Items:**

- Openable items:

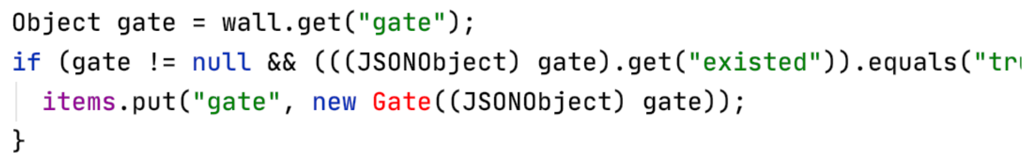
- Adding new openable item (ex: a gate) can be added smoothly, by theses steps:

- let us pretend we have this item details:



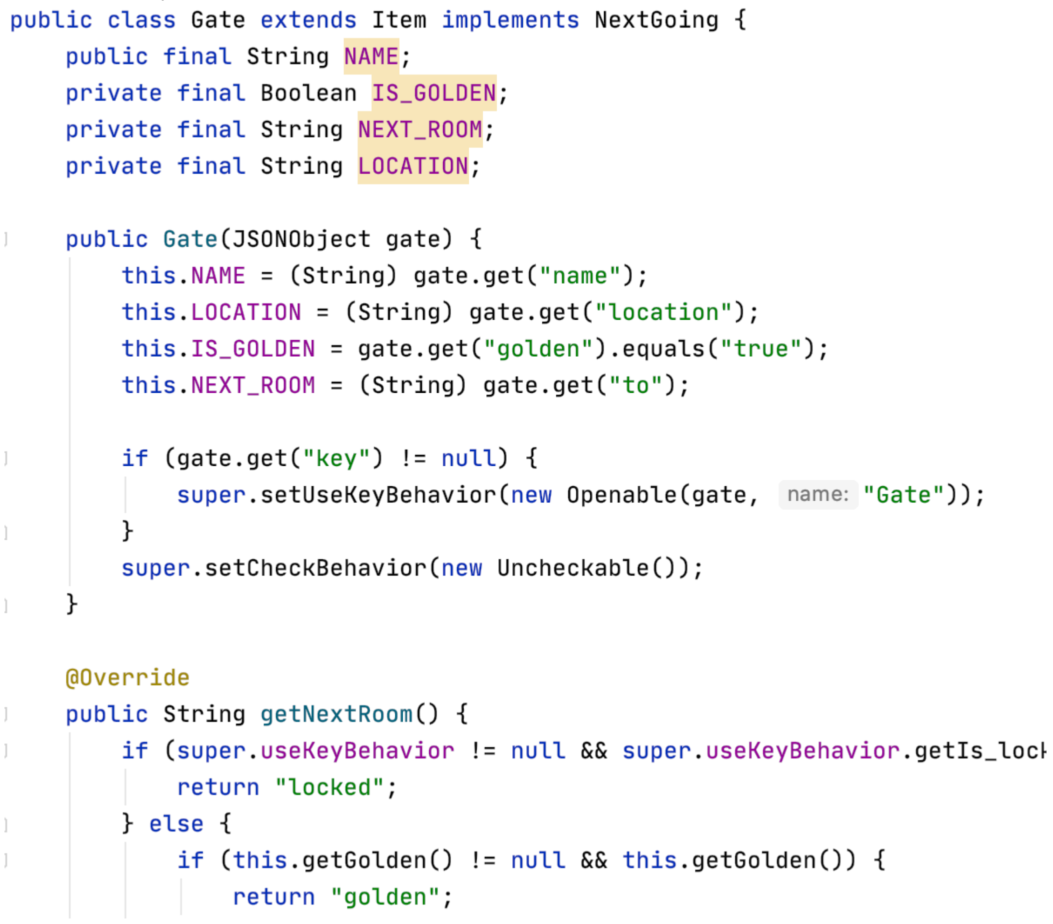
```
"gate": {
  "existed": "true",
  "name": "room_5-gate_1",
  "is_locked": "false",
  "to": "room_2",
  "golden": "false",
  "location": "e5"
}
```

- Adding few new lines in ItemsFactory constructor:



```
Object gate = wall.get("gate");
if (gate != null && (((JSONObject) gate).get("existed")).equals("tr
  items.put("gate", new Gate((JSONObject) gate));
}
```

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



```
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:

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

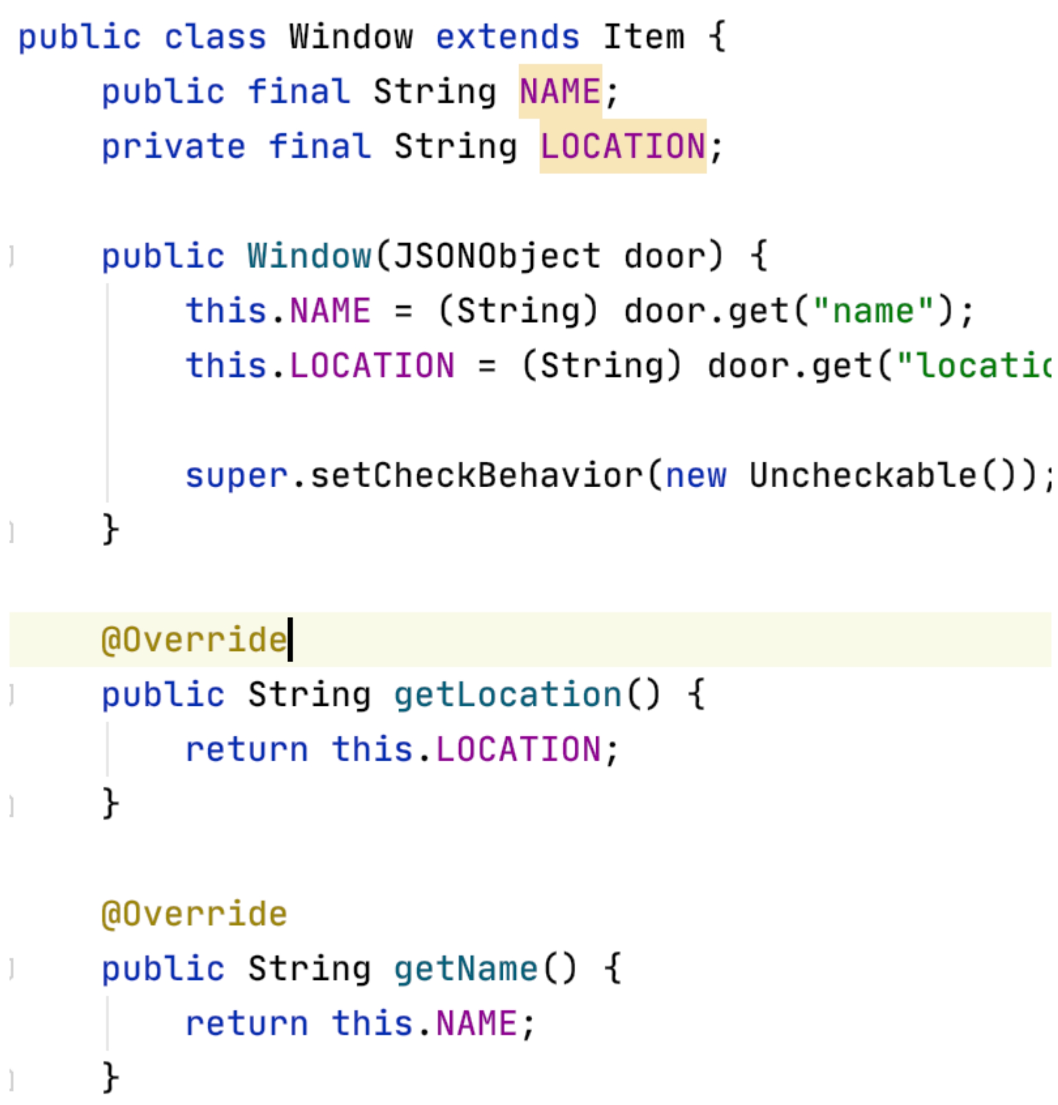
- let us pretend we have this item details:

```
"window": {
  "existed": "true",
  "location": "d1"
}
```

- Adding few new lines in ItemsFactory constructor:

```
Object window = wall.get("window");
if (window != null && (((JSONObject) window).get("existed")).equals("t
  items.put("window", new Window((JSONObject) window));
}
```

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



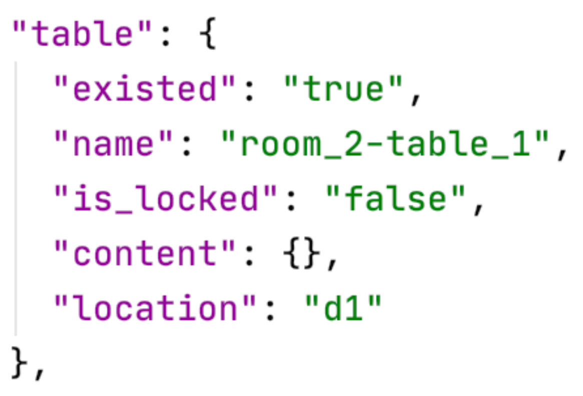
```java
public class Window extends Item {
    public final String NAME;
    private final String LOCATION;

    public Window(JSONObject door) {
        this.NAME = (String) door.get("name");
        this.LOCATION = (String) door.get("locati

        super.setCheckBehavior(new Uncheckable());
    }

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

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

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

- Unlocked_Checkable items:

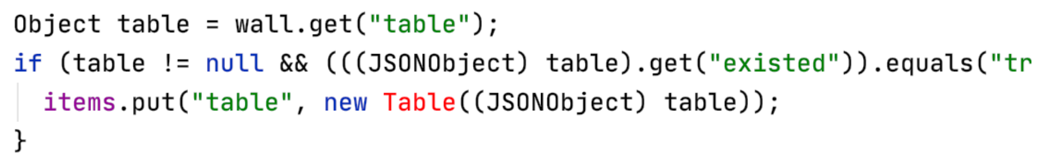
- Adding new unlocked_chackable item (ex: a table) can be added smoothly, by theses steps:

- let us pretend we have this item details:



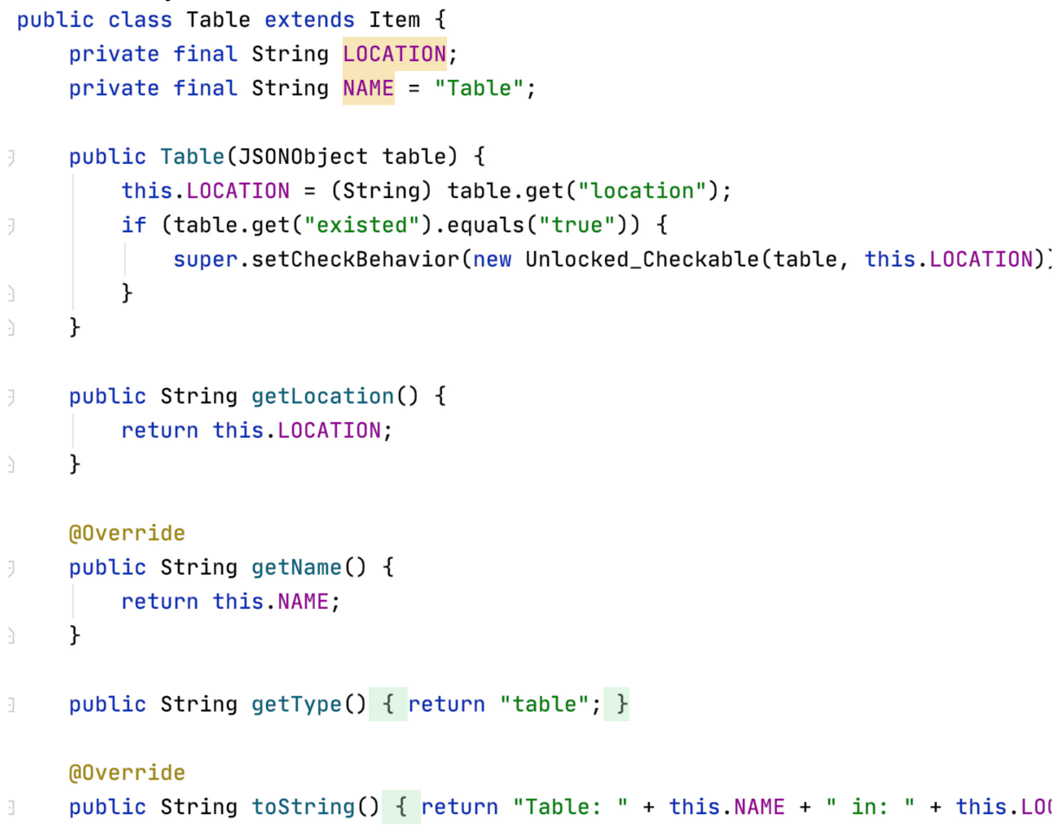
```
"table": {
  "existed": "true",
  "name": "room_2-table_1",
  "is_locked": "false",
  "content": {},
  "location": "d1"
},
```

- Adding few new lines in ItemsFactory constructor:



```
Object table = wall.get("table");
if (table != null && (((JSONObject) table).get("existed")).equals("tr
  items.put("table", new Table((JSONObject) table));
}
```

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



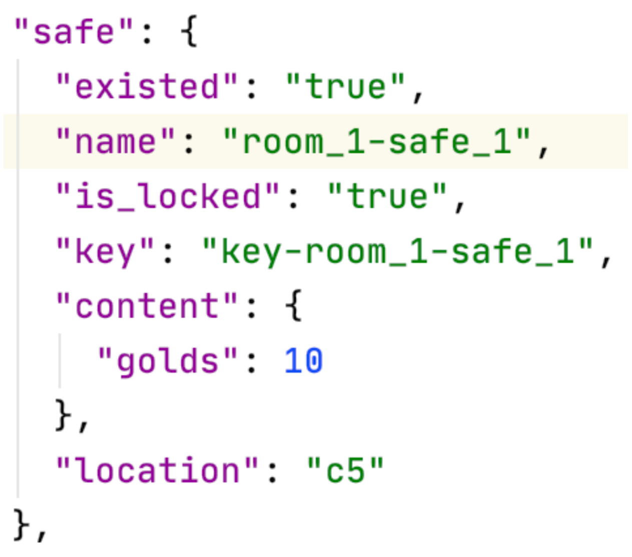
```
public class Table extends Item {
    private final String LOCATION;
    private final String NAME = "Table";

    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:

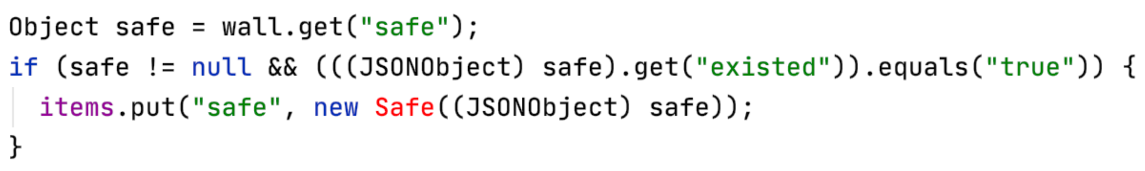
- Adding new openable item (ex: a safe) can be added smoothly, by theses steps:

- let us pretend we have this item details:



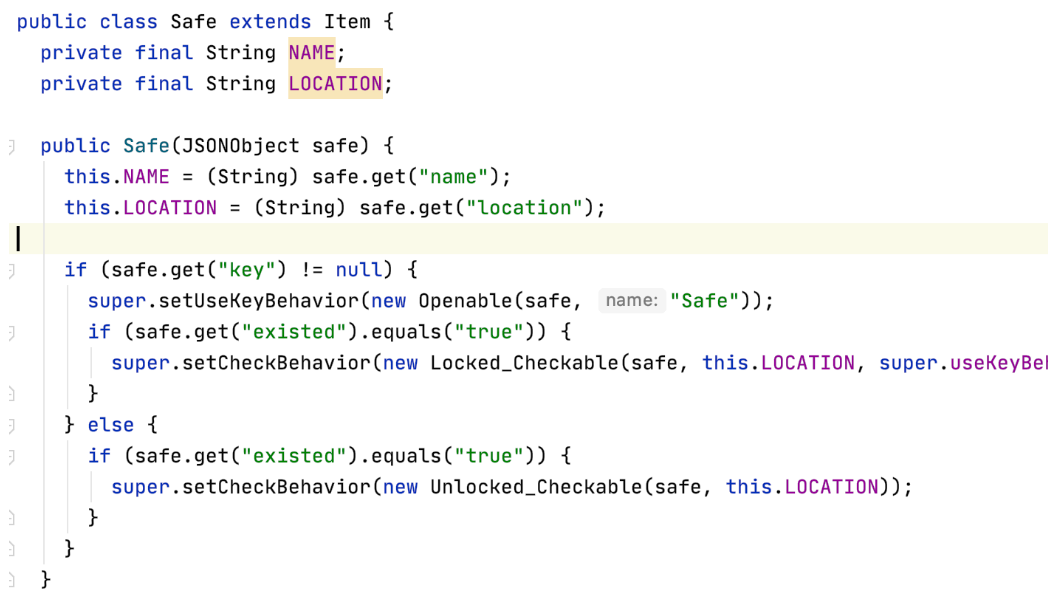
```
"safe": {
  "existed": "true",
  "name": "room_1-safe_1",
  "is_locked": "true",
  "key": "key-room_1-safe_1",
  "content": {
    "golds": 10
  },
  "location": "c5"
},
```

- 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:



```
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 Items are added.