World Navigator

TABLE OF CONTENTS

Abstract

This document will describe and study the requested assignment that revolves around making a world navigator program which complies with some of the clean coding principles.

We will showcase a screenshot of the project itself, as well as some code snapshots.

Section 1: How to Play

The world navigator game is all about exploring a maze, collecting loot, and trading with merchants to reach the main objective.

You have been tasked to collect as much gold as you can to reach a required threshold amount before the time expires.

Check your pockets, you may have some gold at the very beginning of the game.

To start a game, simply select which map you would like to load, each map may differ substantially; as it has different timers, loot, and rooms.

```
12:58:07 | Select Map
12:58:07 | ----------------------------------------
12:58:07 | 1. Default.json
12:58:07 | 2. mymap.json
```

After selecting the desired map, the spawn map will be drawn along with the remaining time and the required gold to finish.

```
13:15:47 | ----------------------------------------
13:15:47 |       ********Door********
13:15:47 |       *                  *
13:15:47 |       *                  *
13:15:47 |      Wall               Wall
13:15:47 |       *                  *
13:15:47 |       *                  *
13:15:47 |       ********Shop********
13:15:47 | ----------------------------------------
13:15:47 | I have 250 seconds to collect 600 gold, better get to it.
13:15:47 | Awaiting command...
```

To view essential information, press the status key (default is T).

```
13:17:21 | Status
13:17:21 | ----------------------------------------
13:17:21 | Gold: 425/600
13:17:21 | ----------------------------------------
13:17:21 | Keys:    Door Keys:  5   10  18
13:17:21 |          Chest Keys: 3   1   4   2
13:17:21 | ----------------------------------------
13:17:21 | Flashlight charge:   40%
13:17:21 | ----------------------------------------
13:17:21 | Remaining time: 207 seconds
13:17:21 | ----------------------------------------
```

Try to meet the gold quota within the time limit.

```
13:25:50 | It's over.
13:25:50 |
13:25:50 | Status
13:25:50 | ----------------------------------------
13:25:50 | Gold: 425/600
13:25:50 | ----------------------------------------
13:25:50 | Keys:    Door Keys:  5   10  18
13:25:50 |          Chest Keys: 3   1   4   2
13:25:50 | ----------------------------------------
13:25:50 | Flashlight charge:   40%
13:25:50 | ----------------------------------------
13:25:50 | Remaining time: 0 seconds
13:25:50 | ----------------------------------------
```

Command	Default Key Binding
Forward	**W**
Right	**D**
Left	**A**
Backward	**S**
Status	**T**
View Map	**M**
Reset Game	**R**
Create Checkpoint	**C**
Load Checkpoint	**L**
Affirmative	**Y**
Negative	**N**

If you do not feel comfortable with a key binding, simply navigate to the main activity and change it, you can also set automated commands instead of manually entering it at the beginning of the game, do note that this will deduct a penalty of five seconds for each command.

```
public static void main(String[] args) {
  // Override default binding controls here, i.e. setForwardKey('8'); (optional)
  setForwardKey('8');

  // Set predefined commands here, use: setBatchControls(getForwardKey(), ...),
  // this will automate the movement of the player and deducts 5 seconds for every move (optional)
  setBatchControls(getForwardKey());

  start();
}
```

Once you collect the required gold, the game will finish.

```
13:44:19 | I think I have more than enough for today
13:44:19 |
13:44:19 | Status
13:44:19 | ----------------------------------------
13:44:19 | Gold: 775/600
13:44:19 | ----------------------------------------
13:44:19 | Keys:    Door Keys:  5   10  18
13:44:19 |          Chest Keys: 3   1   4   2
13:44:19 | ----------------------------------------
13:44:19 | Flashlight charge:   40%
13:44:19 | ----------------------------------------
13:44:19 | Remaining time: 206 seconds
13:44:19 | ----------------------------------------
```

Do note that if you wish to resume the game in another state, you can take advantage of the checkpoint functionality.

To create a checkpoint, press the checkpoint key (default is C).

```
13:51:10 | A checkpoint has been made
```

You can load any checkpoint in any state by pressing the load key (default is L), do not worry if you have saved a checkpoint of a different map, it will load successfully.

```
13:50:37 | ----------------------------------------
13:50:37 | 1. checkpoint.ser
13:50:37 | 2. checkpoint1.ser
13:50:37 | 3. checkpoint2.ser
13:50:37 | 4. checkpoint3.ser
13:50:37 | 5. IWillThrowYouAnException.ser
13:50:37 | 0. Exit
13:50:37 | ----------------------------------------
```

If you happen to be stuck, you can reset the game by pressing the reset game key (default is R), this will return you to the spawn point, resets the countdown timer, and removes any progress that has been made.

```
15:48:13 | Resetting...
15:48:13 | ----------------------------------------
15:48:13 |       ********Door********
15:48:13 |       *                   *
15:48:13 |       *                   *
15:48:13 |      Wall                Wall
15:48:13 |       *                   *
15:48:13 |       *                   *
15:48:13 |       ********Shop********
15:48:13 | ----------------------------------------
15:48:13 | Awaiting command...
```

Section 2: Code Structure

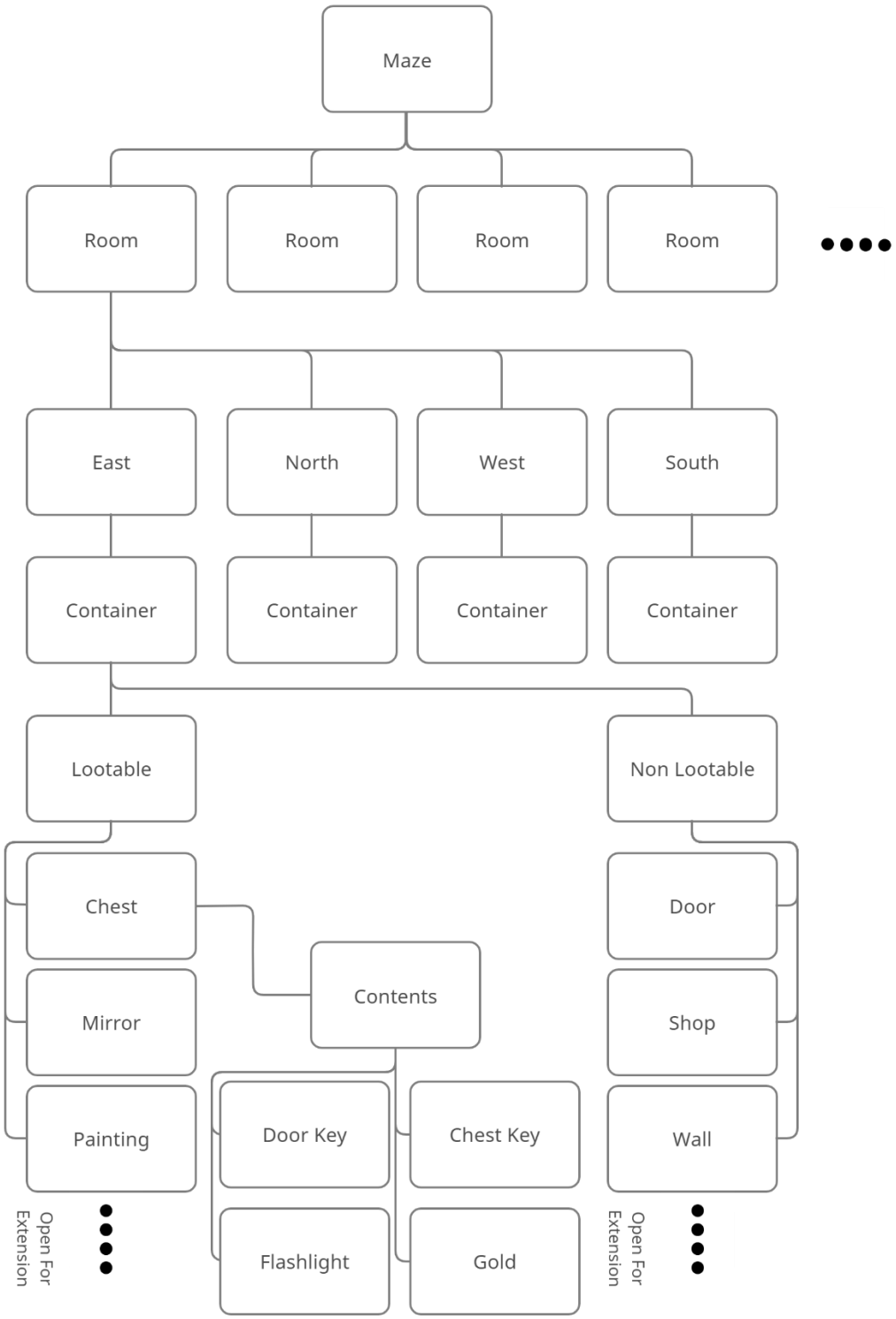


Figure 1: Object Tree of the Program

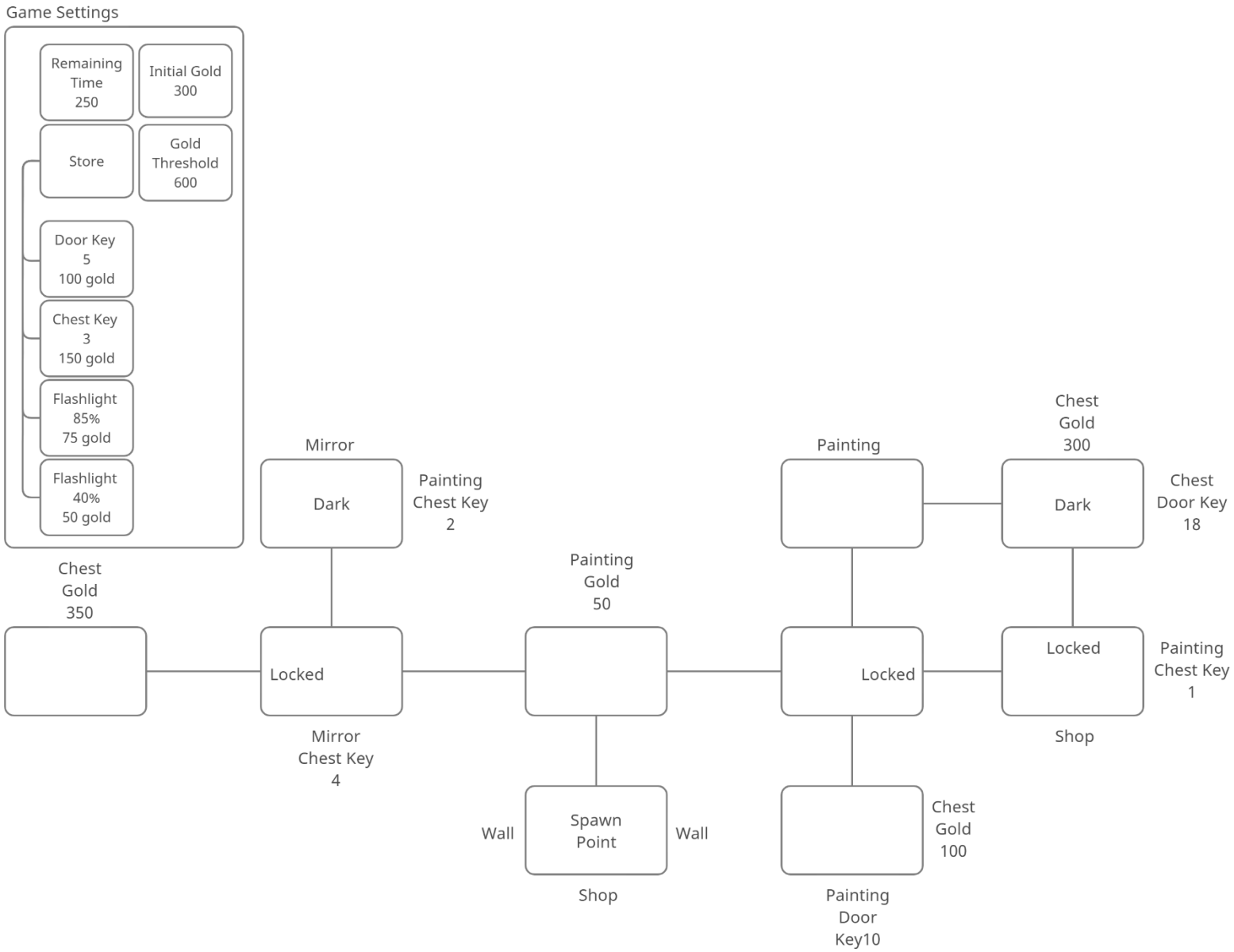
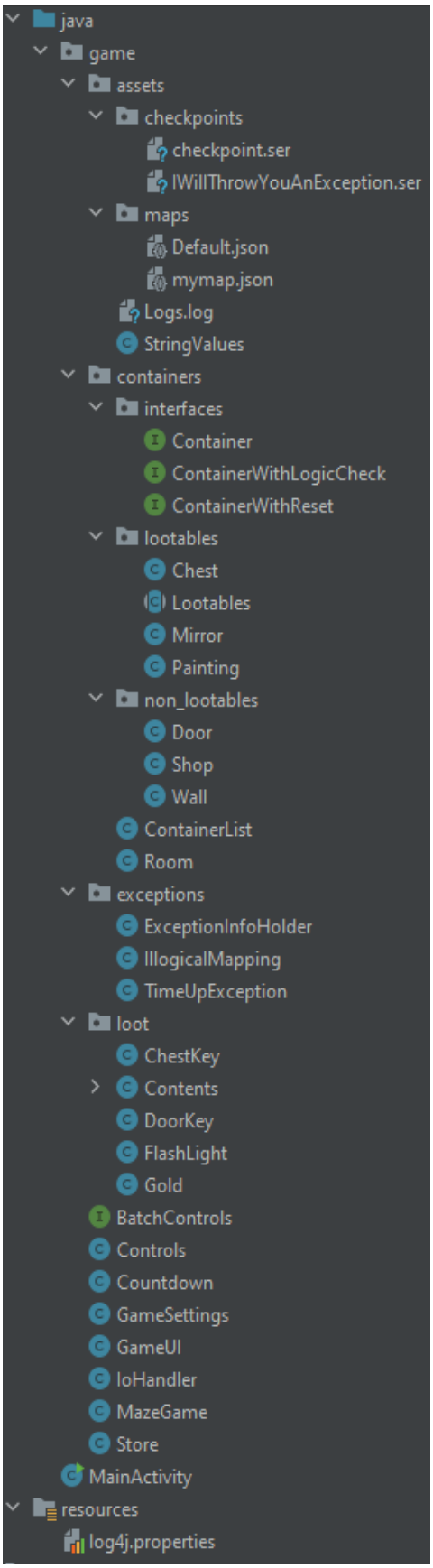


Figure 2: Visualized Default Map



Game: A package which contains all the required classes and assets to operate

- Assets
  - Checkpoints: Stores all saved checkpoints
  - Maps: Store any created map here
  - Log.log: Logs are stored here
  - StringValues: Access/edit dialogues here
- Containers
  - Interfaces: Enforces method implementation
  - Lootables: Lootable containers
  - Non-Lootables: Non-lootable containers
  - ContainerList: binds containers
  - Room: Contains container objects
- Exceptions: Holds exception classes
- Loot: Houses container's contents
- BatchControls: enforces batch implementation
- Controls: Stores controls binding
- Countdown: Handles the timer as a thread
- GameSettings: Contains the deserialized data
- GameUI: Implements methods for the user
- IoHandler: handles logs and console streaming
- MazeGame: The main hub of the game
- Store: Contains the store items

Resources: Stores log4j library properties

Each room must have four containers (each denotes a direction).

Any content "loot" which have a value of 0 or less will be ignored.

The standards of creating a map must be followed when making a JSON file inside the map's directory, any violation will cause an exception to be thrown when deserializing/validating

The guidelines of creating a proper map file are thus:

- The program must have the permission to read the maps directory
- The JSON file must follow the following layout (refer to figure 1 and the default JSON file for clarification):
    - RemainingTime (int)
    - InitialGold (int)
    - GoldThresholdObjective (int)
    - MerchantStore (array of loot with a price entry)
    - Rooms (an array of rooms having containers in each direction and whether the room is dark or not)
    - Each container must have the following entries:
        - Type: denotes if it was a door or a chest for example
        - Which (lootables only): denotes what type of loot is in there (-1 for none)
        - Value (lootables only): denotes the value of the loot (-1 for default)
        - Price (merchant only): denotes the price of the loot
        - To (doors only): denotes which room does this door led to
        - IsLocked (doors only): denotes whether the door is locked or not
- The JSON file must adhere to logical rules, even if the structure is well defined, these logical rules are:
    - The RemainingTime must be at least one minute
    - The InitialGold must not be negative
    - The InitialGold must be less than GoldThresholdObjective
    - The map must have enough gold to ensure winning conditions (startingGold + allContainerGold - StoreItems >= GoldThreshold)
    - The spawn room must not be dark
    - All doors must lead to an existing room
    - All doors must not lead to the same room
    - All doors must have a door in the opposite room
    - All rooms must have at least one door
    - All door keys must correspond to an existing door
    - All locked doors must have a key
    - All chest keys must correspond to an existing chest
    - All chests must have a key

Section 3: Design Patterns Compliance

*The code also satisfies Google's styling guide implicitly since google-java-format is installed; by pressing ctrl+alt+L, IntelliJ automatically reformats the code by google formatting xml file rules

Practice / Design Pattern	Description	Compliant?	Reason
Builders	Used in a situation where there are many constructors with a chance of extending	Compliant	A container may have a variety of contents (loot)
Singleton	Deprive the opportunity of creating an object	Compliant	Having static variables and methods with a private default constructor is a form of singleton pattern
Enforce noninstantiability with a private constructor	Deprive the opportunity of creating an object	Compliant	Many classes have private constructor with static variables and methods
Avoid finalizers	Uncaught exceptions inside a finalizer will not print a warning, it will also cause severe performance penalty	Compliant	Finalizers are not implemented in any package
Always override hashCode when you override equals	Help the class to function properly with all hash-based collections (mainly for objects distinction)	Compliant	All classes which override the equals method also overrides the hashCode method
Always override toString	Makes the class much more pleasant to read	Compliant	All classes (non-abstract) have the toString method overridden (returns all of the interesting information contained in the object)
Minimize the accessibility of classes and members	Hides all of its implementation details	Compliant	Some stringValues's variables use unmodifiableList rather than a mere list, making it immutable
In public classes, use accessor methods, not public fields	Implementing accessor methods (getters) and mutators (setters)	Compliant	All variables are private and can only be accessed with a getter/ setter (if saw fit)
Do not use raw types in new code	Use of raw types lose safety and expressiveness of generics (compiler will not have enough type information to perform all type checks necessary to ensure type safety)	Compliant	All generics are parameterized
Eliminate unchecked warnings	Suppressing warning hides flaws in the code, which will be harder to troubleshoot if an issue arises	Non-Compliant	When deserializing a checkpoint, the compiler will complain about casting from an object to a list, there is no feasible way to avoid it other than suppressing it
Prefer lists to arrays	Lists provides a versatile way to manipulate data, use arrays over lists only when the length is constant	Compliant	Arrays are never used, since there is no preferred implementation for it
Consistently use the Override annotation	Use the Override annotation on every method declaration that you believe to override a super class declaration.	Compliant	Ever overridden method has the annotation
Use varargs judiciously	varargs methods are a convenient way to define methods that require a variable number of arguments, but they should not be overused	Compliant	Batch controls can receive an indefinite number of controls
Return empty arrays or collections, not nulls	There is no reason ever to return null from an array- or collection-valued method instead of returning an empty array or collection	Compliant	Null values are not even used in this project
Prefer for-each loops to traditional for loops	For-each is easier to read and have relatively less coding	Compliant	For-each is implemented, unless a counter is needed, then the for loop would be the better choice
Know and use libraries	Using libraries ensures that the code is written have high standards since experts wrote it	Compliant	Log4j is implemented here, also java.util and java.io
Prefer primitive types to boxed primitives	Primitives are more space and time efficient	Compliant	Primitives are always used when applicable, boxed primitives are used in generics

Practice / Design Pattern	Description	Compliant?	Reason
Avoid Strings where other types are more appropriate	Strings tend to be slower, less flexible and more error-prone	Compliant	String data type is only used to represent data and for serializing
Beware the performance of string concatenation	Using the string concatenation operator repeatedly to concatenate n strings requires time quadratic in n.	Compliant	StringBuilder is used when there are too many concatenations
Adhere to generally accepted naming conventions	Follow typographical and grammatical naming conventions	Compliant	All variable naming complies to those conventions
Use exceptions only for exceptional conditions	Exceptions are used when unwanted events transpire	Compliant	Exception can be thrown when there is a serialization error, or it could be as simple as the timer runs out
Use checked exceptions for recoverable conditions and runtime exceptions for programming errors	checked exceptions: for conditions from which the caller can reasonably be expected to recover unchecked exceptions: should not be caught.	Compliant	Most of the checked exceptions are recoverable, unless if it will impair the rest of the code, fail-fast approach is imposed.
Favor the use of standard exceptions	Easier to read and use because it matches established conventions with which programmers are already familiar	Compliant	User defined exceptions are only made there are no standard exception that fulfills the situation's needs
Do not ignore exceptions	Do not let catch blocks empty.	Non-Compliant	InterruptedException is impossible to be thrown in the main thread, however, it will be logged just in case
Implement Serializable judiciously	Adding implements Serializable is the easiest way to serialize a class, but it decreases the flexibility to change a class's implementation once it has been released.	Non-Compliant	Although serializing Is a bad practice in the long term, however, it is highly unlikely to modify the program. In addition, the value being serialized is non-critical (checkpoints) and justified to be incompatible with any build since the newer version may remove/add a feature, causing inconsistent results
Consider using a custom serialized form	Do not accept the default serialized form without first considering whether it is appropriate.	Compliant	An explicit serial version UID in every serializable class is declared
Solid Principles Compliance			
Single Responsibility Principle	A class should have one, and only one, reason to change.	Partial Compliance	Having only one job for each class may defeat the purpose of encapsulation, there should be a "sweat spot" for decomposing the jobs into sub-jobs
Open Closed Principle	You should be able to extend a classes behavior, without modifying it.	Compliant	Some components, such as containers, can be extended without modifying the base code
Liskov Substitution Principle	Derived classes must be substitutable for their base classes.	Compliant	The base class matches with the specifications of the parent class (a door is a container, a chest is a lootable container)
Interface Segregation Principle	Make fine grained interfaces that are client specific.	Compliant	Each container must implement three methods, however, there are some optional methods that the user can choose to implement or not
Dependency Inversion Principle	Depend on abstractions, not on concretions.	Partial Compliance	Same as the O in the SOLID, the containers have a layer of abstraction with its dependances, but that is not always the case on all of the classes
Sonar List Compliance (All SonarLint rules are respected)			
Cognitive Complexity	Each method should have at most 15 control statements in each method	Compliant	Large method now comprises into smaller, logical methods.
Limit Break/Continue Statements to One Per Loop	Doing so improves code tracking	Compliant	All loops contain at most one break/continue statement
Replace the use of System.out or System.err by a logger	Standard outputs should not be used directly to log anything	Compliant	All output to the console is implicitly logged into a log file
Provide the parametrized type for this generic.	Generic wildcard types should not be used in return types	Compliant	There are no wildcards implemented into the program
Use indentation to denote the code conditionally executed by this "if"	A conditionally executed single line/block should be denoted by indentation	Compliant	Indentation is Implemented

Section 4: Extension Exercise

Suppose that we need a new container called “thief”, which steals gold from the player when interacted.

Requirements:

1. Fill your desired map with the locations where the thief happens to be there, for simplicity, we will be adding two rooms for the sample map, the first one contains one thief, and the other contains two thieves, each thief contains a “deduct” attribute (specifies the amount of stolen gold) which will be decoded later, remember to adhere to the logical rules when creating any map since its very strict.

2. Create a new container class called thief in the subdirectory of “non_lootables” (since the thief should not contain any loot for the player)

```
package game.containers.non_lootables;

public class Thief {
}
```

3. In order for the class to adhere to the container rules, we must implement the container interface (for this example, we will not implement the optional interfaces)

```
public class Thief implements Container {
    @Override
    public String getDisplayName() {
        return null;
    }

    @Override
    public Container handleContainerJSONContents(JSONObject jsonObject) throws FileNotFoundException {
        return null;
    }

    @Override
    public void onInteractListener(Contents inventory) {
    }
}
```

4. Fill the methods in the following logic:
    a. **The first method returns a string which will be shown during the map drawing.** Make the display name return “Chest”; so that the user is tricked into thinking that the container is a chest filled with loot.

    b. **The second method handles the JSON decoding that you previously made, for convenience, the method already gives you the JSON file which contains the child where you specified as the thief.** You should read the JSON and create a new object here

    c. **The third method gets invoked whenever the user interacts with the container during the game.** The stealing process should occur here, you can have any method from the program at your disposal, even though this will result in highly dependent classes, it will improve the flexibility of the containers, such that the container can control anything, for example, depleting the flashlight

```java
public class Thief implements Container {
  boolean isStolen = false;
  int stealingAmount;

  public Thief(int stealingAmount) {
    this.stealingAmount = stealingAmount;
  }

  @Override
  public String getDisplayName() {
    return "Chest";
  }

  @Override
  public Container handleContainerJSONContents(JSONObject jsonObject) {
    return new Thief(((Long) jsonObject.get("deduct")).intValue());
  }

  @Override
  public void onInteractListener(Contents inventory) {
    if (!isStolen) {
      int startingGold = inventory.getGold().getAmount();
      int stolenGold = min(startingGold, stealingAmount);
      if (stolenGold == 0) println("The thief pitied me");
      else println("A thief stole from me " + stolenGold + " gold");
      inventory.getGold().setAmount(startingGold - stolenGold);
      isStolen = true;
    } else println("Speaking of fooling me twice :|");
  }
}
```

5. Even though the class is fully operational, the program must be able to recognize and call it automatically; head to game → containers → ContainerList and add to the “buildContainersList” hash map the following:
    a. A string which denotes the JSON type of container
    b. An object instance of such container

```java
private static Map<String, Container> buildContainersList() {
  Map<String, Container> containersListBuilder = new LinkedHashMap<>();
  containersListBuilder.put(CONTAINER_MERCH, new Shop());
  containersListBuilder.put(CONTAINER_DOOR, new Door( to: -1,  isLocked: false));
  containersListBuilder.put(CONTAINER_PAINTING, new Painting());
  containersListBuilder.put(CONTAINER_MIRROR, new Mirror());
  containersListBuilder.put(CONTAINER_CHEST, new Chest());
  containersListBuilder.put(CONTAINER_WALL, new Wall());
  containersListBuilder.put("Thief", new Thief( stealingAmount: -1));
  return containersListBuilder;
}
```

Note that if you have implemented any optional interfaces, you will have to add into the other hash maps as well.

```
private static Map<String, ContainerWithReset> buildGameResetList() {
  Map<String, ContainerWithReset> gameResetListBuilder = new LinkedHashMap<>();
  gameResetListBuilder.put(CONTAINER_DOOR, (ContainerWithReset) containersList.get(CONTAINER_DOOR));
  gameResetListBuilder.put(CONTAINER_CHEST, (ContainerWithReset) containersList.get(CONTAINER_CHEST));
  //gameResetListBuilder.put("Thief", (ContainerWithReset) containersList.get("Thief"));
  return gameResetListBuilder;
}
```

Observe how the program communicates with the “Thief” class functionality, note that the pseudo chest is here while there is a thief underlying it.

```
16:11:13 | ----------------------------------------
16:11:13 |       ********Door********
16:11:13 |       *                  *
16:11:13 |       *                  *
16:11:13 |      Chest               Shop
16:11:13 |       *                  *
16:11:13 |       *                  *
16:11:13 |       *******Mirror*******
16:11:13 | ----------------------------------------
16:11:13 | I have 60 seconds to collect 1000 gold, better get to it.
16:11:13 | Awaiting command...
t
16:11:14 |
16:11:14 | Status
16:11:14 | ----------------------------------------
16:11:14 | Gold: 800/1000
16:11:14 | ----------------------------------------
16:11:14 | Remaining time: 60 seconds
16:11:14 | ----------------------------------------
16:11:14 | Awaiting command...
16:11:14 | Only one minute left, I still need 200 gold.
a
16:11:15 | A thief stole from me 100 gold
16:11:15 | Awaiting command...
a
16:11:15 | Speaking of fooling me twice :|
16:11:15 | Awaiting command...
t
16:11:16 |
16:11:16 | Status
16:11:16 | ----------------------------------------
16:11:16 | Gold: 700/1000
16:11:16 | ----------------------------------------
16:11:16 | Remaining time: 58 seconds
16:11:16 | ----------------------------------------
16:11:16 | Awaiting command...
```

Section 5: Further Notes

There are a few notes that should be covered before diving into the code.

- Even though the container classes follow the open–closed principle, however, the same thing cannot be said for loot classes, since it is tightly coupled to the logic if the game. For instance, if we added a new loot such as silver, we would have to create a whole new means on how holding such loot would affect the gameplay, i.e., it does not contribute to the winning conditions, but it can be traded, or you can convert it into gold, etc. Also, this will result in interfering with the containers, so that they can interact with such loot.

- All string values have been migrated into the “StringValues” class as a static final variable, this should ease changing any dialog/exception messages.

- Anything which have been printed into the console is already being logged into the “Log.log” file located in the “assets” directory.

- If you wish to change the default controls indefinitely, go to the “Controls” class and change it to your preference. Do note that the user can key bind the controls at the “main”.

- Any loot that has been inserted into the JSON with an invalid value (less than zero) is omitted.

- Containers may throw an exception if you do not comply with their guidelines.

- If you encountered an error code 3001, it may be because of the following reasons:

    - The JSON syntax is invalid.
    - The program received an unexpected datatype, for example, found string while expecting an integer.
    - A new class is created but not added into the hash map.
    - The JSON keys are undefined.
    - A JSON key was expected, but not found.

    You have been given the reason and the line number to troubleshoot the problem

```
17:17:58 | Error decoding the JSON file, ensure that the structure is compliant (error code 3001)
17:17:58 | ----------------------------------------
17:17:58 | Cannot invoke "java.lang.Boolean.booleanValue()" because the return value of "org.json.simple.JSONObject.get(Object)" is null
17:17:58 | Line number: 165
```

- Any modification with the serialized checkpoint will throw an exception
- When setting batch commands, remember to add the map selection as the first argument, then add the rest of the commands “for example, setBatchControls('1', getForwardKey(), getRightKey());”.
- Batch controls only supports chars as an argument; hence, you cannot select maps/checkpoint that correspond to more than a digit.
- The program does not follow all of the requested requirements in the assignment, this includes:
    - **The UI is based on string commands:** it is highly improbable to have a game with commands in a form of string rather than a single character (pressing “W” is far more convenient than pressing “forward”).
    - **Selling to the trader:** doing so could break the game, since when selling an imperative loot such as a key, the user will never be able to open the container anymore.
    - **Silhouette:** you can just print something when interacting with the mirror in the class’s “onInteractListener” method.

```
@Override
public void onInteractListener(Contents inventory) {
  super.onInteractListener(inventory);
  println("I can see myself :)");
}
```

```
17:43:49 | ----------------------------------------
17:43:49 |       ********Door********
17:43:49 |       *                  *
17:43:49 |       *                  *
17:43:49 |      Door               Door
17:43:49 |       *                  *
17:43:49 |       *                  *
17:43:49 |       *******Mirror*******
17:43:49 | ----------------------------------------
17:43:49 | Awaiting command...
s
17:43:50 | Found key for chest/s number
17:43:50 | 4
17:43:50 | I can see myself :)
17:43:50 | Awaiting command...
```

    - **Having an exit door as a winning condition:** changed into collecting gold until a threshold is reached.

- **Concurrency:**
  Since there was a required timer, it is in our best interest to utilize threads; however, there was a corner case issue risen when employing threading.

  When using batch controls to automatically call commands, the timer should catch each command and deduct 5 seconds. However, starting threads may take some time (depending on the scheduler prioritizing mechanism); making the batch controls to continue functioning while the timer is waiting for the scheduler to execute the thread.

  The solution is to simulate an “await” command (similar to the C# language), this can be done by checking the state of the thread, if the thread is terminated (execution is done) and the timer has started, then the rest of the program will resume.

```
private static void startGame() {

  location = getRooms().get(0);
  drawMap(location,  isCharted: false);
  lastKnownLitRoom = location;

  if (!countdown.isStarted()) thread.start();

  while (getInventory().getGold().getAmount() < getGoldThresholdObjective()) {

    // Ensures that the thread is fired, hence, avoiding automated commands stream from colliding
    // with the timer
    if (thread.getState() != Thread.State.TERMINATED) continue;

    println(UI_AWAITING_COMMAND);
```

- **Used data structures:**
  - LinkedHashMap “HashMap”: Needed for a key/value pair (loot and price)
  - HashSet “Set”: ordered, nonrepeatable (comparing chest keys with chests id)
  - ArrayList “List”: A versatile approach to store elements than an array (store keys in the inventory)

- **Status Codes:**
  - Exit code 0: The program ended successfully; winning conditions achieved
  - Exit code 1: The program ended successfully; the time ran out
  - Exit code 3000: Error fetching the JSON file
  - Exit code 3001: Error decoding the JSON file
  - Exit code 3002: Illogical JSON mapping