Lab 4: Interfaces

Instruction

1. Click the provided link on CourseVille to create your own repository.
2. Open IntelliJ and then “Create a project” and set project name in this format **Progmeth_Lab4_2023_2_{ID}_{FIRSTNAME}**
   Example: Progmeth_Lab4_2023_2_6631234521_Samatcha.

3. Initialize git in your project directory
   a. Add .gitignore.
   b. Commit and push initial codes to your GitHub repository.

4. Implement all the classes and methods following the details given in the problem (some files or part of the files are already given, please see the src folder in the given file) statement file which you can download from CourseVille.
   a. You should create commits with meaningful messages when you finish each part of your program.
   b. Don’t wait until you finish all features to create a commit.
5. Test your codes with the provided JUnit test cases, they are inside package test.grader
   a. If you want to create your own test cases, please put them inside package test.student
   b. Aside from passing all test cases, your program must be able to run properly without any runtime errors.

6. After finishing the program, create a UML diagram and put the result image (UML.png) at the root of your project folder.
7. Export your project into a jar file called Lab4_2023_2_{ID} and place it at the root directory of your project. Your jar file must contain source code. Your jar file must be on GitHub too.
   a. Example: Lab4_2023_2_6631234521.jar
8. Push all other commits to your GitHub repository.

1. Problem Statement

"Board Fight" a board game is like a monopoly, but instead of buying assets, you will need to collect resources and fight each other.

Objective:

Survive through 40 turns and be the last player standing.

Game Setup:

1. Two players (P1 and P2) start with a Stick in their inventory.
2. Players take turns to roll a dice, move on the board, and interact with tables and randomizers.

Board Layout:

1. The board consists of 9 positions numbered from 0 to 8.
2. Tables are placed at positions 2 and 6, and randomizers at positions 0, 4, and 8.

Game Elements:

Player:

Each player has a name, health points (HP), and inventory.
HP must be kept above 0 to stay in the game.

Inventory:

Initially contains a Stick as a weapon for each player.
Players can collect items from randomizers.

Tables:

There are two types of tables: Oven and Enchanter.
Players can interact with tables using items from their inventory.

- Oven : players can use "Raw" items in your inventory with an oven to cook them.
  * If players use a "Cooked" item with an oven, it will change to "Burnt".
- Enchanter : player can use any item that has level with an enchanter to upgrade their level.

Randomizer:

At positions 0, 4, and 8 on the board.
Players automatically interact with randomizers to get a random item.

Turns:

Players take turns, rolling a dice to determine their movement on the board.
Movement is modulo 9, ensuring a circular board.

Player's Turn:

Roll Dice:

Players roll a dice (1 to 6) to determine their movement.

Move on the Board:

Players move forward based on the dice roll.
The board is circular, so if the position exceeds 8, it wraps around. (Note that moving around the board is circular, but the attackable range of weapons is not circular.

Interact with Board Elements:

Check if the player lands on a randomizer or a table.
If on a randomizer, receive a random item automatically.
If on a table, interact with the table by selecting items from the inventory to use with the table.

Use Inventory:

Players can use items from their inventory.
Weapons can be used against other players, and non-weapon items have various effects e.g. healing, attack buff and wearing armor.

End Turn:

Attack buff effects decrease, and the turn counter increases.

Winning the Game:

The game ends after 40 turns.
The player with the highest remaining HP is the winner.
If multiple players have the same highest HP, they all win.

Game Flow Example:

Players take turns rolling the dice, moving on the board, and interacting with randomizers and tables.
They use items from their inventory strategically to gain advantages or attack opponents.
The game continues until the 40 turns are completed.

Game Over:

The game ends when 40 turns are completed or only one alive player remains.
The winner or winners with the highest remaining HP are declared.

2. Implementation Details

To complete this assignment, you need to understand about Interfaces.

*** Note that Access Modifier Notations can be listed below**

+ (public)
# (protected)
- (private)
underlined (static)
ALL_CAPS (final variable)
italic (abstract)

* Also note that while other methods have been provided, only methods that you have to implement are listed here.

2.1 Package item /*PARTIALLY PROVIDED*/

This package contains all types of item, including base class (abstract class) and interface for items.

2.2.1 Package item.base /*ALREADY PROVIDED*/

This package contains all abstract classes of items. All classes are completely provided, but you should know some information about them.

*2.2.1.1 Abstract Class **BaseItem***

This class is a generalized class of all items in the game.

Fields

Name	**Description**
- String name	Name of item

Methods

Method	**Description**
+ BaseItem(String name)	Constructor
+ void setName(String name)	Set name of the item unless name is blank string, set name to “Unnamed Item”
+ String getName()	Getter of name

*2.2.1.2 Abstract Class **BaseEquipment** extends **BaseItem***

This class is a subclass of BaseItem, it's a generalized class of all items that can be used multiple times before break.

Fields

Name	**Description**
- int durability	Number of time that item can be use

Methods

Method	**Description**
+ BaseEquipment(String name, int durability)	Constructor
+ void setDurability(int durability)	Set the item's durability unless it's negative, set to 0.
+ int getDurability()	Return durability of item.

*2.2.1.3 Abstract Class **BaseWeapon** extends **BaseEquipment***

This class is a subclass of BaseEquipment. It's a generalized class of all weapons that can be used to attack other players.

Fields

Name	**Description**
- int RANGE	Range that weapon can attack - 0 means it can attack only players that are in the same position, 2 means it can attack players in the same position and 2 blocks to the left and right (total 5 blocks, or less than that if it close to edge of board). - Range cannot be a negative number

Methods

Method	**Description**
+ BaseWeapon(String name, int durability, int range)	Constructor
+ int getRange()	Return range of item
+ abstract int getAtt()	Return attack damage of weapon
+ String toString()	Return a string in this format <name> (Att: <att>, Range: <range>, <durability> uses left) Example: "Stick (Att: 1, Range: 0, 3 uses left)"

*2.2.1.5 Abstract Class **BaseArmor** extends **BaseEquipment***

This class is a subclass of BaseEquipment. It's a generalized class of all armor type in the game that will reduce the damage being attack

Methods

Method	**Description**
+ BaseArmor(String name, int durability)	Constructor
+ int abstract getDef()	Return the defense value
+ String toString()	Return a string in this format <name> (Def: <Def>) Example: "Boots (Def: 20)"

*2.2.1.5 Abstract Class **BaseConsumption** extends **BaseItem***

This class is a subclass of BaseItem. It's a generalized class of all consumable items. There is no specific field or method in this class, but it is a good practice to create this generalized class for possible further update or maintenance (not in this lab).

Methods

Method	**Description**
+ BaseConsumption(String name)	Constructor

2.2.2 Package item.usage **/*ALREADY PROVIDED*/**

This package contains all interface for items and a necessary enum.

2.2.2.1 Enum **CookState**

This enum is for an item that can be cooked at the Oven. There are RAW, COOKED and BURNT.

2.2.2.2 Interface **Cookable**

This interface is for an item that can be cooked at the Oven. Items must have listed methods below.

Methods

Method	**Description**
CookState getCookState()	Return a cookstate.
void setCookState(CookState cookState)	Set a cookstate.

2.2.2.3 Interface **Upgradable**

This interface is for an item that can be upgraded at the enchanter. Items must have listed methods below.

Methods

Method	**Description**
int getLevel()	Return a current item's level.
void setLevel(int level)	Set a item's level.
int getMaxLevel()	Return a max level of the item.

2.2.2.4 Interface **Healable**

This interface is for an item that can be used to recover a player's hp. Items must have listed methods below.

Methods

Method	**Description**
int getRecoverPoint()	Return a healing amount of the item.

2.2.2.5 Interface **AttBuffable**

This interface is for an item that can be used to increase attack damage for a temporary turns. Items must have listed methods below.

Methods

Method	**Description**
int getAttBuff()	Return an amount of damage that will be increased.
void getBuffTurn()	Return an amount of turn that buff will affect.

2.2.3 Package item.armor /*PARTIALLY PROVIDED*/

This package contains all armor types.

2.2.3.1 Class **Boots** extends **BaseArmor**

/*You need to implement this class from scratch*/

Fields

Name	**Description**
- int DEF	Defense value of Boots.

Methods

Method	**Description**
+ Boots()	Constructor Set name to “Boots”, durability = 5 and DEF = 1.
+ int getDef()	Return DEF

2.2.3.2 Class **Suit** extends **BaseArmor**

Implements: Upgradable

/*You need to implement this class from scratch*/

Fields

Name	**Description**
- int level	Current level of Suit.
- int MAX_LEVEL	Max level of Suit.
- int[] DEF	Defense value of Suit for each level (0 to MAX)

Methods

Method	**Description**
+ Suit()	Constructor Set name to “Suit”, durability = 10 Set level to 0, max level to 3 initialize DEF to { 1,2,3,5 }

+ int getDef()	Return DEF of current level.
+ int getLevel()	Return current level
+ void setLevel(int level)	Set the current level. If level is out of proper range [0, max] set to 0.
+ int getMaxLevel()	Return MAX_LEVEL
+ String toString()	Return a string in this format <name> (Def: <def>, Level: <level>) Example: “Suit (Def: 999, Level: 200)”

2.2.3.3 Class **Helmet** extends **BaseArmor** **/*ALREADY PROVIDED*/**

Fields

Name	**Description**
- int DEF	Defense value of Helmet.

Methods

Method	**Description**
+ Helmet()	Constructor Set name to “Helmet”, durability = 5 and DEF = 1.
+ int getDef()	Return DEF

2.2.4 Package item.weapon /*PARTIALLY PROVIDED*/

This package contains all weapon types.

2.2.4.1 Class **Stick** extends **BaseWeapon**

/*You need to implement this class from scratch*/

Fields

Name	**Description**
- int ATT	Attack damage of Stick.

Methods

Method	**Description**
+ Stick()	Constructor Set name to “Stick”, durability = 3, range = 0 and ATT = 1.
+ int getAtt()	Return ATT

2.2.4.2 Class **Sword** extends **BaseWeapon**

Implements: Upgradable

/*You need to implement this class from scratch*/

Fields

Name	**Description**
- int level	Current level of Sword.
- int MAX_LEVEL	Max level of Sword.
- int[] ATT	Attack damage value of Sword for each level (0 to MAX).

Methods

Method	**Description**
+ Sword()	Constructor Set name to “Sword”, durability = 15 and range = 1. Set level to 0, max level to 3 initialize ATT to { 3,5,8,12 }
+ int getAtt()	Return ATT of current level.
+ void setLevel(int level)	Set the current level. If level is out of proper range [0, max] set to 0.
+ String toString()	Return a string in this format <name> (Att: <att>, Range: <range>, Level: <level>, <dur> uses left) Example: “Sword (Att: 223, Range: 100, Level: 99, 99 uses left)”
+ int getLevel()	Return the current level.
+ int getMaxLevel()	Return the max level.

2.2.4.3 Class **Bow** extends **BaseWeapon**

Implements: Upgradable

/*ALREADY PROVIDED*/

*All fields name and type and methods are the same as class Sword, Just some fields' values listed below are changed.

- name = “Bow”
- durability = 10
- ATT = { 2,3,5,8 }
- range = 2

2.2.5 Package item.consumption /*PARTIALLY PROVIDED*/

This package contains all consumption item types, i.e.potion and food.

2.2.5.1 Class **Pork** extends **BaseConsumption**

Implements: Cookable, Healable

/*You need to implement this class from scratch*/

Fields

Name	**Description**
- CookState cookState	Cook state of Pork.
- HashMap<CookState, Integer> RECOVER_PT	Amount of healing for each cook state.

Methods

Method	**Description**
+ Pork()	Constructor Set name to “Pork” Set cook state to RAW. Initialize the hash map RECOVER_PT, map from all cook states to recovery value as (RAW -> 1, COOKED -> 2 and BURNT -> 0) ***Documentation for HashMap at the last section.*
+ int getRecoverPoint()	Return an amount of healing by checking the cook state.
+ String toString()	Return a String in this format <Capitalized cook state> <item name> (+ <recovery point> HP) example : “Cooked Pork (+2 HP)”
+ Getter and Setter for cookState	

2.2.5.2 Class **GoldenApple** extends **BaseConsumption**

Implements: Cookable, Healable

/*ALREADY PROVIDED*/

*All fields name and type and methods are the same as class Pork, Just some fields' values listed below are changed.

- name = “GoldenApple”
- RECOVER_PT = { RAW -> 2, COOKED -> 5, BURNT -> 0 }

2.2.5.3 Class **Pill** extends **BaseConsumption**

Implements: Healable

/*You need to implement this class from scratch*/

Fields

Name	**Description**
- int RECOVER_PT	Amount of healing

Methods

Method	**Description**
+ Pill()	Constructor Set name to “Pill” and RECOVER_PT = 2
+ String toString()	Return a string in this format <name> (+<recover_pt> HP) Example: “Pill (+2 HP)”
+ getRecovePoint()	Return the recover_pt.

2.2.5.4 Class **HealingPotion** extends **BaseConsumption**

Implements: Healable, Upgradable

/*You need to implement this class from scratch*/

Fields

Name	**Description**
- int level	Current level of HealingPotion.
- int MAX_LEVEL	Max level of HealingPotion.
- int[] RECOVER_PT	Amount of healing for each level.

Methods

Method	**Description**
+ HealingPotion()	Constructor Set name to “HealingPotion”. Set level to 0, max level to 3. Initialize RECOVER_PT to { 3,5,7,10 }.

+ int getRecoverPoint()	Return an amount of healing by checking the current level.
+ void setLevel(int level)	Set the current level. If level is out of proper range [0, max] set to 0.
+ String toString()	Return a string in this format <name> (+<recover_pt> HP, Level: <level>) Example: “HealingPotion (+3 HP, Level: 0)”
+ int getLevel()	Return the current level.
+ int getMaxLevel()	Return the max level.

2.2.5.5 Class **StrengthPotion** extends **BaseConsumption**

Implements: AttBuffable, Upgradable

/*You need to implement this class from scratch*/

Fields

Name	**Description**
- int level	Current level of StrengthPotion.
- int MAX_LEVEL	Max level of StrengthPotion.
- int[] ATT_BUFF	Amount of damage buff for each level.
- int BUFF_TURN	Amount of turns that buff will affect.

Methods

Method	**Description**
+ StrengthPotion()	Set name to “StrengthPotion”. Set level to 0, max level to 3. Set buff turn to 3. Initialize ATT_BUFF to { 3,5,7,10 }.
+ int getAttBuff()	Return an att buff.
+ int getBuffTurn()	Return buff turn.
+ void setLevel(int level)	Set the current level. If level is out of proper range [0, max] set to 0.

+ String toString()	Return a string in this format <name> (+<att_buff> for next <buff_turn> turns, Level: <level>) Example: “StrengthPotion (+3 Att for next 3 turns, Level: 0)”
+ int getLevel()	Return the current level.
+ int getMaxLevel()	Return the max level.

2.2.5.6 Class **UltimatePotion** extends **BaseConsumption**

Implements: Healable, AttBuffable

/*ALREADY PROVIDED*/

Fields

Name	**Description**
- int ATT_BUFF	Amount of damage buff.
- int BUFF_TURN	Amount of turns that buff will affect.
- int RECOVER_PT	Amount of healing.

Methods

Method	**Description**
+ UltimatePotion()	Set name to “UltimatePotion”. Set att buff to 5. Set buff turn to 3. Set recover pt to 5.
+ String toString()	Return a string in this format <name> (+<att_buff> for next <buff_turn> turns, +<recover_pt> HP) Example: “UltimatePotion (+99 Att for next 33 turns, +35 HP)”
+ getAttBuff(), getBuffTurn() and getRecoverPoint()	

2.2 Package player /*PARTIALLY PROVIDED*/

This package contains class Player, Inventory and necessary Exception.

2.2.1 Exception **NegativePositionException** extends **Exception** /*ALREADY PROVIDED*/

Should be thrown when trying to set player position to negative value.

2.2.2 Class **Inventory** /*ALREADY PROVIDED*/

This class represents the player's inventory. There are method addItem(BaseItem item) ,removeItem(BaseItem item) to handle adding and removing items from inventory of a player. **DO NOT** call getItems() and modify an array of items directly.

2.2.3 Class **Player** /*PARTIALLY PROVIDED*/

This class represent player that playing on the board

Fields /*Already provided*/

Name	**Description**
- String name	Player's name.
- int hp	Current hp of the player.
- int MAX_HP	Max hp of the player.
- int pos	Position of the player on board.
- Inventory INVENTORY	Player's inventory.
- Helmet helmetSlot	Player's helmet (wearing).
- Suit suitSlot	Player's suit (wearing).
- Boots bootsSlot	Player's boots (wearing).
- int attBuffing	Amount of attack damage buff.
- int buffRemainingTurn	Remaining turn of attack damage buff.

Methods /*Partially provided*/

Method	Description
+ Player(String name)	/*Already provided*/ Constructor Set name to the given parameter. Set pos to 0. Set MAX_HP to 20 and set hp to MAX_HP. initialize inventory. Set attBuffing and buffRemainingTurn to 0.
+ String useWeapon(BaseWeapon item, Player opponent)	/*Fill Code*/ If the opponent is in an attackable range (see the BaseWeapon documentation), 1. Calculate an attack damage from the weapon and player's buff using calculateAtt(weapon). 2. Call opponent.beingAttack(calculatedValue). 3. Call decreaseWeaponDur(weapon) 4. return a string "<player name> attacked <opponent name>", example: "player1 attacked player2". If opponent is not in a attackable range, return "<opponent name> is not in attackable range" example: "p1 is not in attackable range".
+ void beingAttack(int att)	/*Fill Code*/ Decrease player's hp by att - calculateDef() (if it's negative, decrease 0 hp). Also, call decreaseArmorDur().
+ void decreaseArmorDur()	/*Fill Code*/ Decrease all wearing armor's durability. If durability goes to 0, set that slot to null.

+ void decreaseWeaponDur(BaseWeapon item)	/*Fill Code*/ Decrease the weapon's durability. If durability goes to 0, remove the item from the player's inventory.
+ String useNonWeaponItem()	/*Already provided*/ 1. Initialize an empty ArrayList of string. /*Fill Code*/ 2. Check instances of the item. - If it's a kind of BaseArmor, cast item to type BaseArmor and call wear(item) and add string that return from wear() to array - If it's a kind of Healable, cast item to type Healable and call useHeal(item) and add string that return from useHeal() to array - If it's a kind of AttBuffable, cast item to type AttBuffable and call useBuff(item) and add string that return from useBuff() to array 3. Remove the item from the player's inventory. ***Note that some items could be multiple kinds at the same time, it should call all methods that type is satisfied .** /*Already provided*/ return string that concatenates all strings in the array with "\n". If the array is empty, return "Unknown Item".
- String useHeal(Healable item)	/*Already provided*/ Handle all types of useNonWeaponItem().
- String useBuff(AttBuffable item)	
- String wear(BaseArmor item)	
+ int calculateAtt(BaseWeapon weapon)	/*Already provided*/ Calculate final attack damage.
+ int calculateDef()	/*Already provided*/

	Calculate total defense.
+ void setName(String name)	/*Already provided*/ Set name of the player. If it's an empty string, set it to “Steve”.
+ void setHp(int hp)	/*Already provided*/ Set player’s hp. If it’s out of range [0,MAX_HP], set it to the nearest bound.
+ void setPos(int pos) throws NegativePosException	/*Already provided*/ Set player’s position. If it's negative, throw a NegativePosException.
+ void setAttBuffing(int attBuffing)	/*Already provided*/ Set player’s attack damage buff, If it’s negative, set to 0.
+ void setBuffRemainingTurn(int turn)	/*Already provided*/ Set buff remaining turns. If it’s negative, set to 0. Also, if it’s set to 0, call setAttBuffing(0).
+ void printEquipment()	/*Already provided*/
+ Getters and setters for the remaining field. /*Already provided*/	

2.3 Package table /*PARTIALLY PROVIDED*/

This package contains all types of table (it will appear on board, players can interact with them).

2.2.1 Abstract Class *BaseTable* /*ALREADY PROVIDED*/

This class is a generalized class of tables in the game that players can use an item to interact with.

2.2.2 Class **Oven** extends ***BaseTable*** /*PARTIALLY PROVIDED*/

This class is an oven subclass of BaseTable, players can use a cookable item to interact with.

Methods

Method	**Description**
+ Oven()	Constructor /*Already provided*/
+ String interact(BaseItem baseItem)	If baseItem is a cookable item, then 1. If it's raw, set it to cooked and return the string "Cooking succeed". 2. If it's cooked, set it to burnt and return the string "Your food has been burnt". 3. If it's burnt, return the string "Your food is already burnt, cannot be cooked anymore". If it's not a cookable item, then return the string "This item cannot be cooked".

2.2.3 Class **Enchanter** extends ***BaseTable*** /*PARTIALLY PROVIDED*/

This class is an enchanter subclass of BaseTable, players can use an upgradable item to interact with.

Methods

Method	**Description**
+ Enchanter()	Constructor /*Already provided*/
+ String interact(BaseItem baseItem)	If baseItem is a upgradable item, then 1. If it's not reached max level yet, add 1 level to the item and return the string "Upgrade successfully". 2. If it's already max level, return the string "Already max level". If it's not an upgradable item, then return the string "This item cannot be upgraded".

2.2.4 Class **Randomizer** /*ALREADY PROVIDED*/

This class is a randomizer, players can interact with to get a new random item. (The method interact() has no parameter item to interact with, so it's not a subclass of BaseTable.

2.2.5 Class **ItemRandomWeight** /*ALREADY PROVIDED*/

This class tells a randomizer to know the random weight of each item.

2.4 Package main /*ALREADY PROVIDED*/

2.4.1 Class **Board**

This class represents the game's board and handles game stages.

2.4.2 Class **Main**

This class runs the program.

3. Criteria

There are JUnit tests (52 points) and UML (8 points) for a total of 60 points.

3.1 JUnit Test (52 points)

3.1.1 Class Boots (1 point)

a. testConstructor 1

3.1.2 Class Suit (6 points)

a. testConstructor 1
b. testSetLevel 2
c. testGetDef 2
d. testToString 1

3.1.3 Class Stick (1 point)

a. testConstructor 1

3.1.4 Class Sword (6 points)

a. testConstructor 1
b. testSetLevel 2
c. testGetAtt 2
d. testToString 1

3.1.5 Class Pork (4 points)

a. testConstructor 1
b. testGetRecoverPoint 2
c. testToString 1

3.1.6 Class Pill (2 points)

a. testConstructor 1
b. testToString 1

3.1.7 Class HealingPotion (6 points)

a. testConstructor 1
b. testSetLevel 2
c. testGetRecoverPoint 2
d. testToString 1

3.1.8 Class StrengthPotion (6 points)

a. testConstructor 1
b. testSetLevel 2
c. testGetAttBuff 2
d. testToString 1

3.1.9 Class Player (12 points)

a. testUseWeapon 3
b. testBeingAttack 2
c. testDecreaseArmorDur 2
d. testDecreaseWeaponDur 2
e. testUseNonWeaponItemConsumption 1.5
f. testUseNonWeaponItemEquipAmor 1.5

3.1.10 Class Oven (4 points)

a. testInteract 3
b. testBadInteract 1

3.1.11 Class Enchanter (4 points)

a. testInteract 3
b. testBadInteract 1

3.2 UML (8 points)

You will export a UML only package item and attach it to the repository.

a. Has all classes 2 (-1 for each missing classes)
b. Correct inheritance 3 (-1 for each wrong inheritance)
c. Correct implementing interfaces 3 (-1 for each wrong implementation)

4. Documentation

Class HashMap<K,V>

just like unordered_map in python.

Type Parameters:

K - the type of **keys** maintained by this map

V - the type of mapped **values**

example: HashMap<int, int>

Constructor (initialize):

just use **new HashMap()** to initialize an empty HashMap.

Add value of key: put(key, value)

For example, if you have a

```
private HashMap<int,int> hashMap;
```

and you want to map from 1 to 2, just use

```
hashmap.put(1,2);
```

**If the key already has a value, it will replace the existing value with a new value.*

Get value of key: get(key)

For example, from the above example

```
int x = hashmap.get(1)
```

x will store the value 2

**If the key is not in the keySet, get(key) will return null.*

Full documentation (for other methods and examples)

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html