Lab 4: Interfaces

Instruction

1. Click the provided link on CourseVille to create your own repository.
2. Open Eclipse and then “File > new > Java Project” and set project name in this format **2110215_Lab4_2022_1_{ID}_{FIRSTNAME}**
    - Example: **2110215_Lab4_2022_1_6431234521_Kevin**.
3. Initialize git in your project directory
    - **Add .gitignore.**
    - Commit and push initial codes to your GitHub repository.
4. Implement all the classes and methods following the details given in the problem statement file which you can download from CourseVille.
    - You should create commits with meaningful messages when you finish each part of your program.
    - 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**
    - If you want to create your own test cases, please put them inside package **test.student**
    - Aside from passing all test cases, your program must be able to run properly without any runtime errors.
    - There will be additional test cases to test your code after you submit the final version, **make sure you follow the specifications in this document**.
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_2022_1_{ID}** and place it at the root directory of your project.
    - Example: **Lab4_2022_1_6431234521.jar**
8. Push all other commits to your GitHub repository

1. Problem Statement : Overdue! All You Can Rush

This lab is based on **Overcooked!** game series by **Team17 Digital**, heavily modified to be playable in command line style.

The basic rule of this game is that; you are given a limited kitchen space, and ingredients. You need to serve all the customers' order before the time run out.

1.1 Demonstration

Upon start up the game, you will be greeted by the starting screen, as shown below.

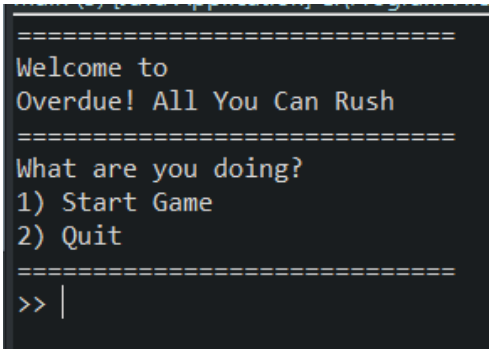

Fig 1. Starting Screen

After the game has been started, you will be taken into the main Kitchen screen. Which looks like below.

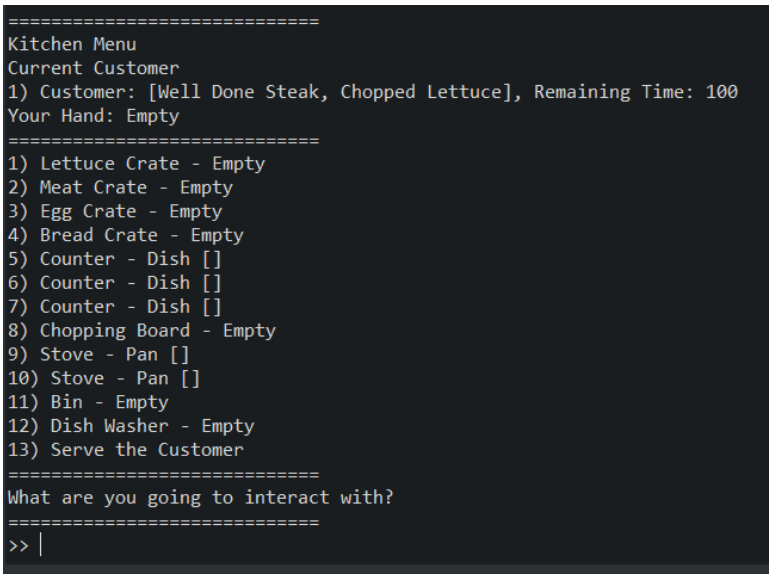

Fig 2. Main Kitchen Screen.

From this screen, you will be able to see the Customer List on the top, denoting their Order, as well as the remaining time. This time will tick down every time you perform an action. Each Customer has different decrement rate. The goal is to serve all the Customer's Order before their time runs out.

To serve a Customer's order, you need to put everything they ordered into the same Dish, hold onto that, and pick serve option. Upon served, the Dish will be cleared and becomes dirty, in which you need to put it into Dish Washer.

To interact with all Kitchen equipment, you simply need to pick said equipment of choice. If your hand is not empty, you will place that object down onto the chosen equipment instead (given that said equipment accept the item that was in your hand). The details for each interaction will be explained later in this document.

2. Implementation Details:

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

This assignment gives you more freedom over how you can implement the solution. There are **more than one** possible ways that the final class diagram could look like, so we will **not** provide a class diagram.

There are **four** packages in the provided files: `application`, `entity`, `logic` and `test`.

You will be implementing most of the class in the `entity` package (Only a few classes are provided, you need to implement the rest from scratch)

There are some test cases given in package `test.grader`. These will help test your code whether it will be able to run or not. However, **some conditions are not tested** in these test cases. If you need to test more conditions, please create your own test case in `test.student` package. However, **it is optional, and won't be graded**.

You can define any additional number of <u>private</u> (but not public, protected or package) fields and methods in addition to the fields and methods specified below. You are encouraged to try to group your logic into private methods to **reduce duplicate code as much as possible**.

Do note that only relevant methods related to logic will be explained.

** Noted that Access Modifier Notations are listed below*

+ (public)
# (protected)
- (private)
<u>Underline</u> (static)
Italic (abstract)

2.1 package `logic`

This package's content is already provided for you. **You do NOT need to edit anything in this package to complete this assignment**. However, you might need to use these methods to help with your implementation.

2.1.1 Class `InvalidIngredientException extends Exception`

A special exception that will be used for a method in the upcoming `LogicUtil`.
You will need to handle this exception as well, more detail later in the document.

2.1.1.1 Constructor

+ InvalidIngredientException(String str)	Initialize the exception object with the message `str`.

2.1.2 Class `LogicUtil`

This class provides a lot of useful methods for the game. You only need to use one, though.

2.1.2.1 Methods

<u>+ Ingredient createIngredientFromName(String s) throws InvalidIngredientException</u>	Returns the specified `Ingredient` object from the input String. It throws `InvalidIngredientException` if the provided name is invalid.

2.1.3 Class `Player`

This class represent the player.

2.1.3.1 Fields and Constructors

- Item holdingItem	A field represent the Item that the player is holding. It can be `null` if player holds nothing.
+ Player()	Initialize a player object. The player holds nothing at the start.

2.1.3.2 Methods

+ boolean isHandEmpty()	Returns **true** if the player's hand is empty
+ Item placeItem()	Make player's hand empty and return the `Item` that player had been holding.
getter/ setter for holdingItem	

2.1.4 Class `StringUtil`

This class provides a useful method for formatting String that you will need in the future.

2.1.4.1 Methods

+ String formatNamePercentage(String name, int percentage)	Returns a `String` in the following format: "Name (percentage%)" – Ex: "Dish (100%)"

2.2 package `entity.base`

2.2.1 Abstract Class `Item`

This class is the base class of anything that a player can hold onto.

You do NOT need to modify this class to complete the assignment.

2.2.1.1 Constructor

# Item(String name)	Initialize the `Item` with the provided name.

2.2.1.2 Methods

+ boolean equals(Object obj)	This method has been specifically written so that 1. If the other object is `Item` type, it will compare both items' respective name together. 2. If the other object is `String` type, it will instead compare the current name with the other `String`.
+ String toString()	Returns the name of the object

2.2.2 Abstract Class `Ingredient extends Item`

This class represents the Ingredient that the player can use to cook into dishes.

You do NOT need to modify this class to complete the assignment.

2.2.2.1 Field and Constructor

- boolean isEdible	A field denote if the `Ingredient` is edible or not
# Ingredient(String name)	Initialize the `Ingredient` with the provided name and set it to be inedible by default.

2.2.2.2 Methods

getter/setter for isEdible	

2.2.3 Abstract Class `Container extends Item`

This class represents the Container type Item that the player can hold onto. This type of Item can hold a certain number of Ingredients inside.

You do NOT need to modify this class to complete the assignment.

2.2.3.1 Fields and Constructor

- ArrayList<Ingredient> content	The current content of the `Container`
- int maxCapacity	How many item can be stored in this `Container`.
- int capacity	How many empty slot left for this `Container`.
# Container(String name, int capacity)	Initialize the `Container` fields with respective values.

2.2.3.2 Methods

+ boolean verifyContent(Ingredient i)	A method used for verifying if the `Ingredient` can be put in this `Container` or not. Its behavior varies between different type of `Containers`.
+ boolean addContent(Ingredient i)	Returns **true** if the specified `Ingredient` can be added into `Container` successfully, returns **false** otherwise. The `Ingredient` can only be successfully added if 1. The `Ingredient` can be put in this container (checked using `verifyContent` method above) 2. The current `Container` has a space for it.
+ void clearContent()	Clear the content of the `Container`
+ void transferContent(Container c)	Transfer all the content from `Container c` Please see the code for more details.
Remaining getter/setters for the fields	

2.2.4 Interface `Choppable`

This interface defines methods for Ingredient that can be chopped.

2.2.4.1 Method

+ *void chop()*	This method will be called when the `Ingredient` got chopped. More information in the upcoming section.
+ *boolean isChopped()*	Self-explanatory

2.2.5 Interface `Cookable`

This interface defines methods for Ingredient that can be cooked.

2.2.5.1 Method

+ *void cook()*	This method will be called when the `Ingredient` got cooked. More information in the upcoming section.
+ *boolean isBurnt()*	Self-explanatory

2.2.6 Interface `Updatable`

This interface defines methods for the object that can update every game step.

2.2.6.1 Method

+ *void update()*	Updates the information. This method is called every end of game step.

2.3 package `entity.ingredient`

This package contains implementation of the **concrete** Ingredients. All classes must be created from scratch.

2.3.1 Class `Lettuce extends Ingredient`

Implements: Choppable

This class represents a Lettuce type Ingredient, which can only be chopped.

2.3.1.1 Fields and Constructor

- boolean chopState	Keeps track if the `Lettuce` has been chopped or not.
+ Lettuce()	Initialize the object with the name "Lettuce" It sets to edible, and not been chopped by default.

2.3.1.1 Methods

+ void chop()	Do nothing if the `Lettuce` has been chopped, otherwise change the state to chopped and the name to "Chopped Lettuce"
+ boolean isChopped()	Returns `chopState`

2.3.2 Class `Egg extends Ingredient`

Implements: Cookable

This class represents an Egg type Ingredient, which can only be cooked.

2.3.2.1 Fields and Constructor

- int cookedPercentage	Keeps track of how "cooked" the `Egg` is.
+ Egg()	Initialize the object with the name "Egg" Sets the `cookedPercentage` to 0

2.3.2.1 Methods

+ void cook()	Increase the `cookedPercentage` by 12 Depending on `cookedPercentage`, change the name and the `isEdible` status to the following; **0 < x <= 50:** Name = "Raw Egg", it is inedible. **50< x <=80:** Name = "Sunny Side Egg", it is edible. **80< x <= 100:** Name = "Fried Egg", it is edible. **x > 100:** Name = "Burnt Egg", it is inedible.

+ boolean isBurnt()	Returns **true** if `cookedPercentage` > 100
+ String toString()	Returns a formatted string in the format of "<`name`> (<`cookedPercentage`>%)" Ex: "Egg (0%)" Please call a method from **`StringUtil`**
Remaining getter/setter	

2.3.3 Class `Meat extends Ingredient`

Implements: Choppable, Cookable

This class represents a Meat type Ingredient, which can be both be chopped and cooked.

2.3.3.1 Fields and Constructor

- boolean chopState	Keeps track if the `Meat` has been chopped or not.
- int cookedPercentage	Keeps track of how "cooked" the `Meat` is.
+ Meat()	Initialize the object with the name "Meat" It sets to not been chopped by default, and sets the `cookedPercentage` to 0

2.3.3.1 Methods

+ void chop()	Do nothing if the `Meat` has been chopped or cooked, otherwise change the state to chopped and the name to "Minced Meat"
+ boolean isChopped()	Returns `chopState`
+ void cook()	Depending on if the meat has been chopped or not, if it's not Increase the `cookedPercentage` by 10 Depending on `cookedPercentage`, change the name and the `isEdible` status to the following; **0 < x <= 50:** Name = "Raw Meat", it is inedible. **50< x <=80:** Name = "Medium Rare Steak", it is edible. **80< x <=** 100: Name = "Well Done Steak", it is edible. **x > 100:** Name = "Burnt Steak", it is inedible.

	If the meat has been chopped, Increase the `cookedPercentage` by 15 Depending on `cookedPercentage`, change the name and the `isEdible` status to the following; **0 < x <= 80:** Name = "Raw Burger", it is inedible. **80< x <= 100:** Name = "Cooked Burger", it is edible. **x > 100:** Name = "Burnt Burger", it is inedible.
+ boolean isBurnt()	Returns **true** if `cookedPercentage` > 100
+ String toString()	Returns a formatted string in the format of "<`name`> (<`cookedPercentage`>%)" Ex: "Meat (0%)" Please call a method from **`StringUtil`**

2.4 package `entity.container`

This package contains implementation of the **concrete** Containers. All classes must be created from scratch.

2.4.1 Class `Dish extends Container`

This class represents a Dish type Container. It can contain four edible Ingredients at maximum, and is necessary to put Ingredients in before serving the customer.

2.4.1.1 Fields and Constructor

- int dirty	The dirtiness of the dish.
+ Dish()	Initialize the object with the name "Dish", with the capacity of 4. Sets `dirty` to 0
+ Dish(int dirty)	Initialize the object with the name "Dish", with the capacity of 4. Sets `dirty` to the specified amount

2.4.2.1 Methods

+ boolean isDirty()	Returns **true** if `dirty` > 0
+ boolean verifyContent(Ingredient i)	Returns **true** if the `Dish` is not dirty and the `Ingredient` is edible

+ void setDirty(int dirty)	Set `dirty` value, **if `dirty` is less than 0, set it to 0** Also, if `dirty` is greater than 0, set the name to "Dirty Dish", otherwise change the name back to "Dish".
+ void clean(int amount)	Clean the Dish; reduce the `dirty` by the specified amount
+ String toString()	If the Dish is dirty, returns a formatted string in the format of "<`name`> (<`dirty`>%)" Ex: "Dirty Dish (0%)" Please call a method from **`StringUtil`** Otherwise, **call toString of the superclass**
Remaining getter/setters	

2.4.2 Class `Pan extends Container`

This class represents a Pan type Container. It can contain one Cookable Ingredients at maximum

2.4.2.1 Constructor

+ Pan ()	Initialize the object with the name "Pan", with the `capacity` of 1.

2.4.2.1 Methods

+ boolean verifyContent(Ingredient i)	Returns true if the `Ingredient` is `Cookable` Hint: **instanceof** can be used to check if the object has implemented certain interface or not as well
+ void cook()	Call the cook method of all the `Ingredients` inside, **only if the Pan is not empty** Hint: You can cast the `Ingredient` to `Cookable` in order to access their specific methods

2.5 package `entity.counter`

This package contains implementation of all the Kitchen equipment. There is one base class given, all other classes must be created from scratch.

2.5.1 Class `Counter`

This class represents the Kitchen counter. The player can put any Item they are holding onto the counter, and said item can be picked up by the player if their hand is empty.

This class is given, you do not need to modify anything in this class. Although you will need to use these details in the future implementation.

2.5.1.1 Fields and Constructor

- String name	Name of the object, mainly used for displaying
- Item placedContent	The `Item` that has been placed on the counter, can be `null` if empty
+ Counter()	Initialize the object with the name “Counter”
+ Counter(Item content)	Initialize the object with the name “Counter” and set the `placedContent` to the given Item

2.5.1.2 Methods

+ boolean isPlacedContentEmpty()	Returns **true** if `placedContent` is `null`
+ void interact(Player p)	**The intended default behavior when the Player interact with the `Counter`** If `placedContent` is empty, then the player place down the item from their hand. Otherwise, it performs a check for each `Item` type and react accordingly. See the code for more details.
+ String toString()	Formatting the text properly, mostly used for display
Remaining getter/setters	

2.5.2 Class `Bin extends Counter`

This class represents the Bin. The player can discard the holding Ingredient, or empty the Container they are holding using this object.

2.5.2.1 Constructor

+ Bin()	Initialize the object with the name “Bin”

2.5.2.2 Methods

+ void interact(Player p)	If the player’s hand is empty, do nothing. Otherwise, check the type of the `Item` that player is holding. If it is `Ingredient`, empty the player’s hand. If it is `Container`, empty said Container instead.

2.5.3 Class `Crate extends Counter`

This class represents the Ingredient Crates. The player can obtain infinite supply of Ingredient if they interact with this object while having empty hand.

2.5.3.1 Fields and Constructor

- String myIngredient	The `Ingredient` that this `Crate` contains, **note that it is stored in the form of `String`**
+ Crate(String s)	Initialize the object with the name “<`Ingredient`> Crate” (Example: “Lettuce Crate” if the parameter is “Lettuce”) and set `myIngredient` properly

2.5.3.2 Methods

+ void interact(Player p)	If the player’s hand is not empty, or there is a content placed on this `Counter`, **call the base Counter behavior.** Otherwise, set the Item that player holds to the new `Ingredient` object created using `createIngredientFromName` from `LogicUtil`.

	Please note that `createIngredientFromName` **can produce exceptions** in case that the given Ingredient name does not exists. In such case, please set the player's hand to `null`. (**Do not printStackTrace, as that will only just shown the error and not correct the behavior**)
Remaining getter/setters	

2.5.4 Class `ChoppingBoard extends Counter`

This class represents the Chopping Board. The player can place any Ingredient onto this object (But not the Container). However, if said Ingredient is Choppable, then it gets chopped as well.

2.5.4.1 Constructor

+ ChoppingBoard()	Initialize the object with the name "Chopping Board"

2.5.4.2 Methods

+ void interact(Player p)	If there is a content placed on the `Counter`, **call the default Counter behavior.** Otherwise, check if the player is holding an `Ingredient` or not. If it is, **call the default Counter behavior**, and call the `chop()` method as well if said `Ingredient` is `Choppable`.

2.5.5 Class `Stove extends Counter`

implements Updatable

This class represents the Stove. The player can put only the Pan on it. This class also updates every player movement step in the game loop, slowly cooking the Ingredient that has been put in the Pan.

2.5.5.1 Fields and Constructor

+ Stove()	Initialize the object with the name "Stove"
+ Stove(Item content)	Initialize the object with the name "Stove" and set the `placedContent` to the `Item` specified

2.5.5.2 Methods

+ void interact(Player p)	If the `placedContent` is not empty, **call the default Counter behavior.** Otherwise, check if the `Item` the player holds is a `Pan` or not. **Call the default Counter behavior** if it is.
+ void update()	If the `placedContent` is a `Pan`, call the `cook()` method from the `Pan` itself.

2.5.6 Class `DishWasher extends Counter`

implements Updatable

This class represents the Dish Washer. The player can put only the Dirty Dish on it. This class also updates every player movement step in the game loop, slowly cleaning the Dish.

2.5.6.1 Fields and Constructor

+ DishWasher()	Initialize the object with the name “Dish Washer”

2.5.6.2 Methods

+ void interact(Player p)	If the `placedContent` is not empty, **call the default Counter behavior**. Otherwise, check if the `Item` the player holds is a `Dish` and is dirty or not. **Call the default Counter behavior** if it is.
+ void update()	If the `placedContent` is a `Dish`, call the `clean()` method with the amount of 15 from the `Dish` itself.

Criteria

test.solution.ingredient.EggTest		3.5
	testConstructor	0.5
	testCook	2
	testIsBurnt	0.5
	testToString	0.5
test.solution.ingredient.LettuceTest		1.5
	testConstructor	0.5
	testChop	1
test.solution.ingredient.MeatTest		6
	testConstructor	0.5
	testChop	0.5
	testCookNormal	2
	testCookChop	2
	testIsBurnt	0.5
	testToString	0.25
	testToStringChopped	0.25

test.solution.container.DishTest		6
	testNoParameterConstructor	0.25
	testParameterConstructor	0.25
	testVerifyContent	1
	testVerifyContentInedible	1
	testVerifyContentDirty	1
	testIsDirty	0.5
	testClean	1
	testToString	0.5
	testToStringDirty	0.5
test.solution.container.PanTest		3.5
	testConstructor	0.5
	testVerifyContent	1
	testVerifyContentNonCookable	1
	testCook	1

test.solution.counter.BinTest		4
	testConstructor	0.5
	testInteract	0.5
	testInteractIngredient	1.5
	testInteractContainer	1.5
test.solution.counter.ChoppingBoardTest		4
	testConstructor	0.5
	testInteract	0.5
	testInteractChoppable	1.5
	testInteractNonIngredient	1.5
test.solution.counter.CrateTest		5.5
	testConstructor	0.5
	testInteract	1.5
	testInteractHandFull	1.5
	testInteractInvalid	2

test.solution.counter.DishWasherTest		6.5
	testConstructor	0.5
	testInteract	0.5
	testInteractNonDish	1.5
	testInteractDish	1.5
	testInteractDirtyDish	1
	testUpdate	1.5
test.solution.counter.StoveTest		5.5
	testConstructor	0.25
	testConstructor1Param	0.25
	testInteract	0.5
	testInteractPan	1.5
	testInteractNonPan	1.5
	testUpdate	1.5
Exception		5
	Handle try/catch	1.5
	Fix the exception properly	1.5
UML		2