

Lab 1: Object-oriented Programming

Instruction

1. Click the provided link on CourseVille to create your own repository.
2. Create a new module in IntelliJ and set module name in this format **2110215_Lab1_2023_1_{ID}_{FIRSTNAME}**
 - Example: **2110215_Lab1_2023_1_6531234521_Samatcha**.
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 (some files or part of the files are already given, please see the src folder).
 - 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.
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 **Lab1_2023_1_{ID}** and place it at the root directory of your project. Your jar file must contain source code.
 - Example: **Lab1_2023_1_6531234521.jar**
8. Push all other commits to your GitHub repository

1. Problem Statement: Item Inventory

As everyone knows, in lots of games there is a feature “Inventory” which is a player storage, carrying player’s items with them to everywhere the player goes to. In this lab, your job is to implement some part of a simple inventory management program. This program can add new items to your game, selling items between players and sell or buy items to market.

The program example is shown below. The program should be run from the Main class in the package main.

```
=====
                Select an option
-----
1. Show market (all items).
2. Show player inventory.
3. Add new items from file.
-----
0. Exit
=====
```

(Main menu)

At the main menu, There are 3 options.

1. Show market (all items)

This option will show all items and their price that exist in the game. There are 10 items that are already provided by default. You also can select “Buy item” to buy items.

```
=====
                Market
-----

1. Sword $100
2. Chestplate $200
3. Boots $150
4. Helmet $150
5. Shield $200
6. Arrow $80
7. Wand $150
8. Apple $10
9. Potion $25
10. Elixir $300

-----
Select an option
1. Buy item
-----
0. Exit Market
=====
```

(Market)

If you choose to buy an item, then you can select which player is going to buy an item, which item you want to buy and how many.

```
=====
Select player to buy item.
-----
1. Vishnu
2. Nattee
-----
0. Back
=====
```

(Select which player is going to buy item)

```
=====
Buy item
-----
1. Sword $100
2. Chestplate $200
3. Boots $150
4. Helmet $150
5. Shield $200
6. Arrow $80
7. Wand $150
8. Apple $10
9. Potion $25
10. Elixir $300
-----
Nattee's money
    $10000
-----
>> Type item's number.
>> Type 0 to back.
=====
```

(Select item to buy.)

```
-----
>> Type item's number.
>> Type 0 to back.
=====
3
=====
>> Type amount.
>> Type 0 to back.
=====
7
=====
```

(Type an amount of item to buy.)

```
7
=====BUY CONFIRMATION=====
    Nattee is buying
    Boots x7 for $1050
>> Type "1" to confirm buying
>> Type anything else to cancel
=====
1
TRANSACTION COMPLETE!
=====
```

(Confirm buying.)

2. Show player Inventory

This option will show a player's inventory (there are 2 inventories in this program). You can see player inventory, sell player's items to other players and to market.

```
=====
Select player to open inventory.
-----
1. Vishnu
2. Nattee
-----
0. Back
=====
```

(Select player to open inventory.)

```
1
=====
      Vishnu's inventory
      $5000
-----

1. Boots $150 x2
-----

Select an option
1. Sell item to another player.
2. Sell item to market.
-----
0. Exit inventory.
=====
```

(player's inventory. You can see items that this player owned.)

- Sell item to another player.

```
1
=====
      Select player to sell to
-----
1. Nattee
-----
0. Back
=====
```

(Select player to sell an item to.)

```
1
=====
      Selling item to Nattee.
           $8950
-----
      Vishnu's inventory
           $5000
-----

1. Boots $150 x2
-----

>> Type item's number to sell
>> Type 0 to back.
=====
```

(Select item to sell.)

```
1
-----
>> Type amount.
>> Type 0 to back.
=====
```

(Type an amount.)

```
1
=====SELL CONFIRMATION=====
      Selling Boots x1 to Nattee
           for $150
>> Type "1" to confirm selling
>> Type anything else to cancel
=====
1
<<TRANSACTION COMPLETE>>
```

(Confirm selling.)

- Sell item to market.

```
=====
Vishnu is selling item to the market.
-----
      Vishnu's inventory
        $5150
-----

1. Boots $150 x1

-----
>> Type item's number to sell
>> Type 0 to back.
=====
```

(Select an item to sell to the market.)

```
1
-----
>> Type amount.
>> Type 0 to back.
=====
1
```

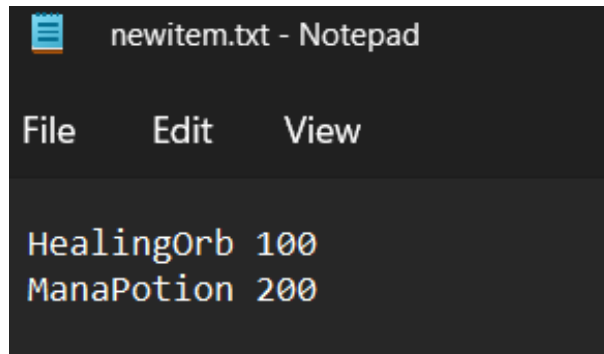
(Type an amount.)

```
1
=====SELL CONFIRMATION=====
      Selling Boots x1 to market
        for $150
>> Type "1" to confirm selling
>> Type anything else to cancel
=====
1
<<TRANSACTION COMPLETE>>
```

(Confirm selling.)

3. Add new items from file.

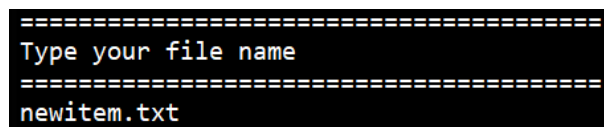
You can add new items using a text file (.txt) in the below format.



```
newitem.txt - Notepad
File Edit View
HealingOrb 100
ManaPotion 200
```

(Text file example.)

When you have entered the menu “Add new items from file” you can type your file name that includes items you want to add.



```
=====
Type your file name
=====
newitem.txt
```

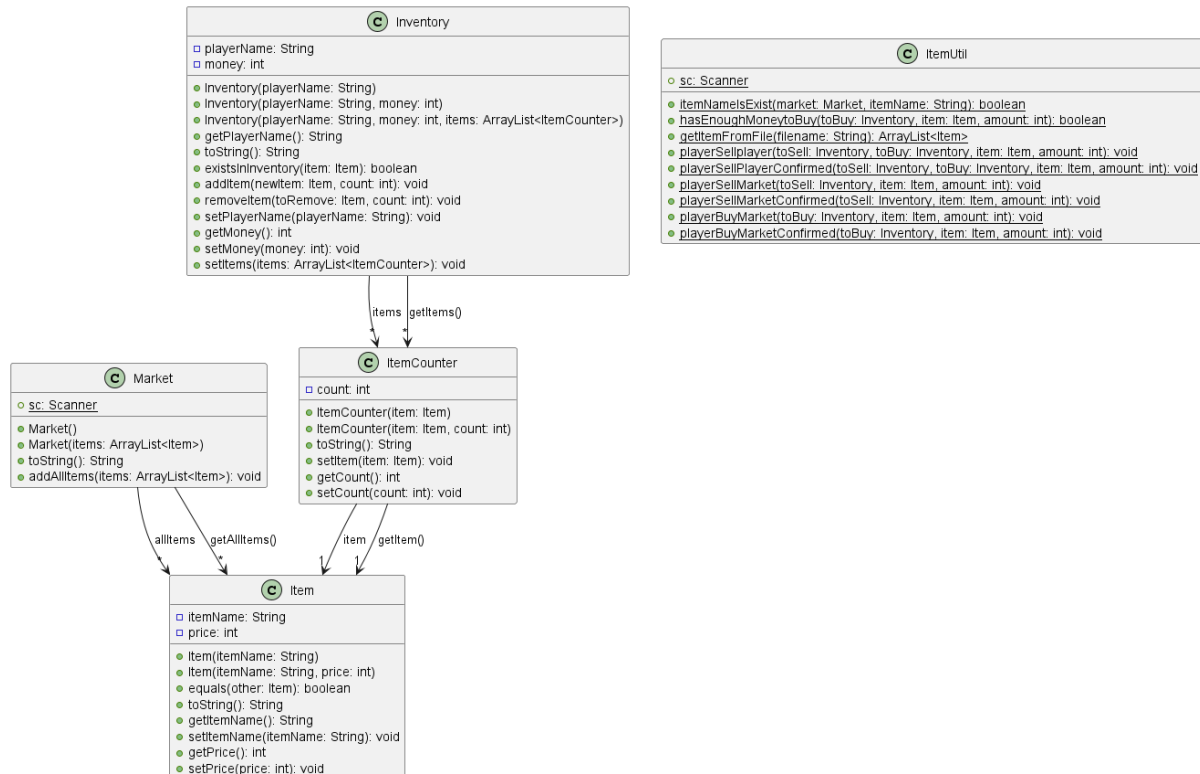
Your new items will appear in the market!



```
=====
Market
-----
1. Sword $100
2. Chestplate $200
3. Boots $150
4. Helmet $150
5. Shield $200
6. Arrow $80
7. Wand $150
8. Apple $10
9. Potion $25
10. Elixir $300
11. HealingOrb $100
12. ManaPotion $200
-----
Select an option
1. Buy item
-----
0. Exit Market
=====
```

2. Implementation Details:

The diagram of the program is illustrated below. There are 5 classes: Item, ItemCounter, Inventory, Market, and ItemUtil.



* Note that Access Modifier Notations can be listed below

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

+ (public)

(protected)

- (private)

2.1 Class Item (package logic)

This class represents an item. Each Item has the item's name and its price.

2.1.1 Fields

- String itemName	Item's name (cannot be blank)
- int price	Item's price (cannot be less than 0)

2.1.2 Constructors

+ Item(String itemName)	/*Fill Code*/ Initialize all fields. (set price to 0) Note: You should use setter to set the value of all fields in order to handle negative value cases and special cases.
+ Item(String itemName, int price)	/*Fill Code*/ Initialize all fields. Note: You should use setter to set the value of all fields in order to handle negative value cases and special cases.

2.1.3 Methods

+ boolean equals(Item other)	/*Fill Code*/ The method checks if this card is the same as the parameter item. Return true if itemName of two items are the same, regardless of their price.
+ String toString()	/*Fill Code*/ Return the string in the pattern {itemName} \${price} example : "Potion \$200"
+ void setItemName(String itemName) throws NameBlankException	/*Fill Code*/ This method sets the item's name, if this method is called with blank string as a parameter, it throws a NameBlankException.

	<p>Hint : You can use method <code>{String}.isBlank()</code> to check if the String contains only just white space.</p> <p>Example, " ".<code>isBlank()</code> will return true.</p>
+ void setPrice(int price)	<p>/*Fill Code*/</p> <p>This method sets the item's price. If the price is below 0, it sets the price to 0 instead.</p>
Getter & Setter methods for all other variables	<p>/*Fill Code*/</p>

2.2 Class ItemCounter (package logic)

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

This class is used to count an amount of an item. For each ItemCounter object. there is an Item that is being counted and an amount of that item.

2.2.1 Fields

- Item item	Item that be counted
- int amount	Item's amount (cannot be less than 0)

2.2.2 Constructor

+ ItemCounter(Item item)	<p>/*Fill Code*/</p> <p>Initialize all fields with the given parameter. (set count to 1)</p> <p>Note: You should use setter to set the value of all fields in order to handle negative value cases and special cases.</p>
+ ItemCounter(Item item,int count)	<p>/*Fill Code*/</p> <p>Initialize all fields with the given parameter. If this constructor is used, the value of this.amount cannot be set to less than 1.</p> <p>Note: You should use setter to set the value of all fields in order to handle negative value cases and special cases.</p>

2.2.3 Methods

+ String toString()	/*Fill Code*/ Return the string in the pattern this.getItem() + " x" + this.getCount() example : "Potion \$200 x4"
Getter & Setter methods for all other variables	/*Fill Code*/ count cannot be less than 0. If the count is less than 0, it will be set as 0.

2.3 Class Inventory (package logic)

This class represents a Player's Inventory. Each Inventory has a name of the player, money, and their items.

2.3.1 Fields

- String playerName	Name of player who is the owner of the inventory (cannot be blank)
- int money	money in the inventory (cannot be less than 0)
- ArrayList<ItemCounter> items	The list of items that counts the number of each item in the inventory.

2.3.2 Constructors

+ Inventory(String playerName)	/*Fill Code*/ Initialize all fields.(set player's name to playerName and set money to 0) Note: You should use setter to set the value of all fields in order to handle negative value cases and special cases.
+ Inventory(String playerName, int money)	/*Fill Code*/ Initialize all fields.(set player's name to playerName and set money to money) Note: You should use setter to set the value of all fields in order to handle negative value cases and special cases.
+ Inventory(String playerName, int money, ArrayList<ItemCounter> items)	/*Fill Code*/ Initialize all fields.(set player's name to

	<p>playerName, set money to money and set ArrayList of items to items)</p> <p>Note: You should use setter to set the value of all fields in order to handle negative value cases and special cases.</p>
--	--

2.3.3 Methods

+ String toString()	Returns the object as a string.
+ boolean existsInInventory(Item item)	<p>/*Fill Code*/</p> <p>This method checks if the given item exists in the inventory. The item exists in the inventory if there is an ItemCounter with a count of 1 or greater in ArrayList items.</p>
+ void addItem(Item newItem, int count)	<p>/*Fill Code*/</p> <p>This method adds new items to the inventory, equal to the given number. If the given count is not a positive integer, then it does nothing.</p> <p>If this item already exists in the inventory, it adds the count to the ItemCounter corresponding to the pre-existing item in ArrayList items.</p> <p>If this item does NOT already exist in the inventory, then initialize a new ItemCounter corresponding to this item with the given count, then add it to ArrayList items.</p>
+ void removeItem(Item toRemove, int count)	<p>This method removes items from the inventory, equal to the given number. If the given count is not a positive integer, it does nothing.</p> <p>If this item exists in the inventory, it decreases the count from the ItemCounter of this existing item in ArrayList items according to the method's count parameter.</p> <p>If the count reaches 0 this way, then the</p>

	itemCounter must also be removed from the list of items.
+ void setPlayerName(String playerName)	/*Fill Code*/ This method will set the name of the player. If the given playerName is empty, set the playerName as "Untitled Player". Hint: There is a non-static method called isBlank() in the String class. It returns true if a String consists of only whitespace. Example: String exampleString = " "; exampleString.isBlank() will return true.
+ void setMoney(int money)	/*Fill Code*/ This method will set money in the inventory. If the money is less than 0, it will be set as 0.
Getter & Setter methods for all other variables	/*Fill Code*/

2.4 Class Market (package logic)

This class contains all of the items that exist in the game (players can own, sell and buy them.).

2.4.1 Fields

- ArrayList<Item> allItems	The list of all items sold in the market.
----------------------------	---

2.4.2 Constructors

+ Market()	/*Fill Code*/ Initialize all fields.
+ Market(ArrayList<Item> items)	/*Fill Code*/ Initialize all fields. Then add All Items from given parameter to the list of all items. Hint: There is a method in this class that you should call.

2.4.3 Methods

+ String toString()	Returns the object as a string.
+ void addAllItems(ArrayList<Item> items)	<p>/*Fill Code*/</p> <p>For each of the items in the given list, if the item already exists in the market, Item will not be added. If there is one more item that has the same name in the list, only the item that comes first in the list will be added. Otherwise, add the item to ArrayList allItems.</p> <p>If the given list is null, do nothing.</p> <p>Print whether the item is added to the market (see example in section 3).</p> <p>Note : The sequence of items that are added to the market have to be the same as the sequence of items in the parameter list, and all of them have to come after items that already exist in the market before.</p>
Getter for allItems	/*Fill Code*/

2.5 Class ItemUtil (package logic)

This class contains some static methods for managing the Item, ItemCounter, Market, and Inventory.

2.5.1 Fields

This class only contains static methods and no fields.

2.5.2 Constructors

This class does not need a constructor.

2.5.3 Methods

+ boolean itemNamelsExist(Market market, String itemName)	<p>/*Fill Code*/</p> <p>This method checks if the given itemName exists in the given Market. Return true if it does, return false otherwise.</p>
---	---

<u>+ boolean hasEnoughMoneytoBuy(Inventory toBuy, Item item, int amount)</u>	<p>/*Fill Code*/</p> <p>This method checks if the given Inventory has enough money to buy the specified amount of item. Return true if it does, return false otherwise.</p>
<u>+ ArrayList<Item> getItemFromFile(String filename)</u>	<p>/*Fill Code*/</p> <p>Three lines of this function have been given to you. Replace “//TODO: Fill Code” with the following algorithm:</p> <p>First, create a scanner to read fileToRead.</p> <p>Then, while the scanner still has lines for it to read, get the string of the line. Split it with " " to get an array, and create a new item like this:</p> <ul style="list-style-type: none"> - The 0th member of the array is the Item Name. (If it's blank, this item will not be added to the list.) - The 1st member of the array is the Price. <p>Finally, return the list of all items created this way . Don't forget to close the scanner. If the file is not found or if there is error while reading the file (for example, the text file isn't properly formatted), this method should return null instead.</p> <p>Hint: You might need to use try catch or throw exceptions here.</p> <p>IMPORTANT: There is no JUnit test for this method. Your program must be able to read an item file during its actual use.</p> <p>Note: Sequence of items in the returning list need to be the same as sequence in the file.</p>
<u>+ void playerSellplayer(Inventory toSell, Inventory toBuy, Item item, int amount)</u>	<p>called when one player wants to sell an amount of item to another player.</p>
<u>+ void playerSellPlayerConfirmed(Inventory toSell, Inventory toBuy, Item item, int</u>	<p>/*Fill Code*/</p> <p>called when the transaction of selling the</p>

<u>amount)</u>	<p>item to another player can be done.</p> <p>The money of the buyer must decrease and the money of the seller must increase according to the price of items.</p> <p>The existence of items in both inventory should be changed.</p>
<u>+ void playerSellMarket(Inventory toSell, Item item, int amount)</u>	called when one player wants to sell an amount of item to the market.
<u>+ void playerSellMarketConfirmed(Inventory toSell, Item item, int amount)</u>	<p>/*Fill Code*/</p> <p>called when the transaction of selling the item to market can be done.</p> <p>The money of the seller must increase according to the price of items.</p> <p>The existence of items in the seller's inventory should be changed.</p>
<u>+ void playerBuyMarket(Inventory toBuy, Item item, int amount)</u>	called when one player wants to buy an amount of item to the market.
<u>+ void playerBuyMarketConfirmed(Inventory toBuy, Item item, int amount)</u>	<p>/*Fill Code*/</p> <p>called when the transaction of buying the item from the market can be done.</p> <p>The money of the buyer must decrease according to the price of items.</p> <p>The existence of items in the buyer's inventory should be changed.</p>

We have already provided these below classes.

2.6 Class NameBlankException (package exception)

Should be thrown when trying to create a new item with a blank name.

2.7 Class Main (package main)

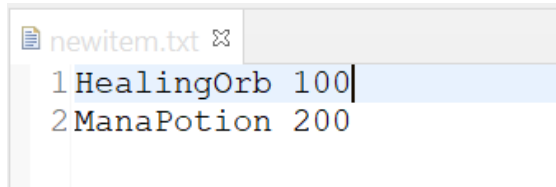
This is where you can run your program.

3. Test Scenario

(User input is in green.)

Since other methods can be tested in the JUnit, we will only display how to test the item import function here. In order to test the import function, we have provided three text files.

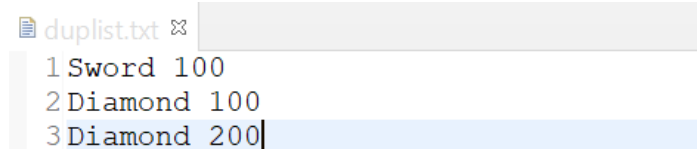
The file “newitem.txt” looks like this.



```
newitem.txt ✕  
1 HealingOrb 100  
2 ManaPotion 200
```

Since this file contains correctly formatted string, when this file is imported, HealingOrb and ManaPotion should be added since they don't already exist.

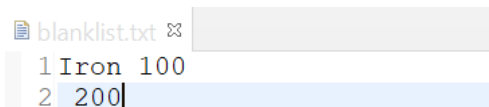
The file “duplist.txt” looks like this.



```
duplist.txt ✕  
1 Sword 100  
2 Diamond 100  
3 Diamond 200
```

Note that the “Sword” already exists and there are two “Diamond” in this text file. It will only add the first “Diamond”.

The file “blanklist.txt” looks like this.



```
blanklist.txt ✕  
1 Iron 100  
2 200
```

The Iron will be the only one item that be added. Since the name of item cannot be blank.

Note that if you import a text file that does not exist, the program shows error message and can continue running.

2110215 Programming Methodology

```
=====
                Select an option
            -----
1. Show market (all items).
2. Show player inventory.
3. Add new items from file.
            -----
0. Exit
=====
1
=====
                Market
            -----

1. Sword $100
2. Chestplate $200
3. Boots $150
4. Helmet $150
5. Shield $200
6. Arrow $80
7. Wand $150
8. Apple $10
9. Potion $25
10. Elixir $100

            -----
Select an option
1. Buy item
            -----
0. Exit Market
=====
0
=====
                Select an option
            -----
1. Show market (all items).
2. Show player inventory.
3. Add new items from file.
            -----
0. Exit
=====
3
=====
Type your file name
=====
newitem.txt
"HealingOrb $100" Added to the market.
"ManaPotion $200" Added to the market.
=====
                Select an option
            -----
1. Show market (all items).
2. Show player inventory.
3. Add new items from file.
            -----
0. Exit
=====
3
=====
Type your file name
=====
duplist.txt
"Sword" is duplicated, Item will not be added.
"Diamond $100" Added to the market.
"Diamond" is duplicated, Item will not be added.
=====
```

2110215 Programming Methodology

```

        Select an option
-----
1. Show market (all items).
2. Show player inventory.
3. Add new items from file.
-----
0. Exit
=====
3
=====
Type your file name
=====
blanklist.txt
Item name cannot be blank! This item will not be added.
"Iron $100" Added to the market.
=====
        Select an option
-----
1. Show market (all items).
2. Show player inventory.
3. Add new items from file.
-----
0. Exit
=====
3
=====
Type your file name
=====
notexist.txt
Cannot find file!
=====
        Select an option
-----
1. Show market (all items).
2. Show player inventory.
3. Add new items from file.
-----
0. Exit
=====
1
=====
        Market
-----

1. Sword $100
2. Chestplate $200
3. Boots $150
4. Helmet $150
5. Shield $200
6. Arrow $80
7. Wand $150
8. Apple $10
9. Potion $25
10. Elixir $100
11. HealingOrb $100
12. ManaPotion $200
13. Diamond $100
14. Iron $100

-----
Select an option
1. Buy item
-----
0. Exit Market
=====
```

4. Criteria

There are a total of 40 points.

4.1 JUnit

InventoryTest

- testConstructor (2)
- testBadConstructor (2)
- testExistsInInventory (1)
- testAddItem (1)
- testSetPlayerName (1)
- testSetMoney (1)
- testSetItems (1)

ItemCounterTest

- testConstructor (2)
- testBadConstructor (2)
- testSetItem (1)
- testSetCount (1)
- testToString (1)

ItemTest

- testConstructor (2)
- testBadConstructor (2)
- testEquals (1)
- testToString (1)
- testSetItemName (1)
- testSetPrice (1)

ItemUtilTest

- testItemNamesExist (1)
- testHasEnoughMoneytoBuy (1)
- testPlayerSellPlayerConfirmed (1)
- testPlayerSellMarketConfirmed (1)
- testPlayerBuyMarketConfirmed (1)

MarketTest

- testConstructor (2)
- testAddAllItems(1)

4.2 Exception (getItemsFromFile Functionality)

File With Valid String Format

- New items are added (1)

File With Duplicate Name

- The items that already exists won't be added (1)
- If There are duplicate name of item in file, only the first item is added (1)

File That Doesn't Exist

- The program shows error message and can continue running (2)
- No new items are added (1)

4.3 UML

- An image file of a correct UML diagram exists in the root of the project (2)