Lab 4: Java Interfaces

Instruction

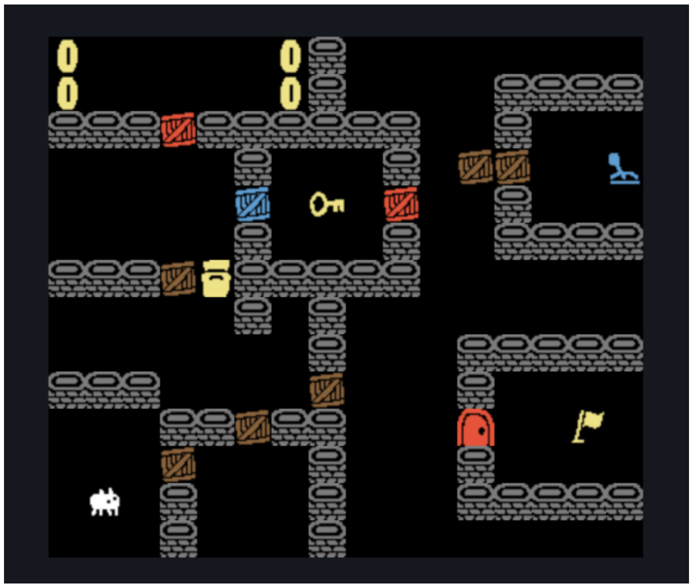
1. Click the provided link on CourseVille to create your own repository.
2. Open Eclipse and then "File > new > Java Project" and set project name in this format **2110215_Lab4_2021_1_{ID}_{FIRSTNAME}**
    - Example: **2110215_Lab4_2021_1_6331234521_Baba**.
3. Initialize git in your project directory
    - **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.
    - **The provided files contain two folders: `src` and `res`. make sure to add both into your project. (And use both of them as Source 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.
    - There will be additional test cases **THAT YOU DO NOT SEE** to test your code after you submit the final version. **Make sure that, not only must your program pass the test cases, it must also follow the specifications in this document closely**.
6. After finishing the program, create a UML diagram using **ObjectAid or any UML drawing tool (create only all interfaces and the classes that you write the code)** and put the result image at the root of your project folder.
7. Export your project into a jar file called **Lab4_2021_1_{ID}** and place it at the root directory of your project.
    - Example: **Lab4_2021_1_6331234521.jar**
8. Push all other commits to your GitHub repository.

1. Problem Statement : Prog Meth is You

You are very famous now, the card game you created is very successful and brings a huge amount of income to the Toy Company. In order to pursue a higher career, you decided to quit the company and become an indie developer.

By sheer coincidence, you come across a veteran indie game developer in a deep mountain. You ask him that you wanted to be trained. However, he said that you must prove that you are worthy by creating a small indie game first.

1.1 Gameplay



This game is partially based on **Baba Is You (2019)** by **Hempuli.** However, the rules and elements are stripped down to basic level to make it easier to understand. You are placed in a 2D Grid where you can walk in 4 cardinal directions (Left-Right-Up-Down). You navigate the maze and can push around boxes (only push!) and make the way to the Flag.

1.2 Prerequisite

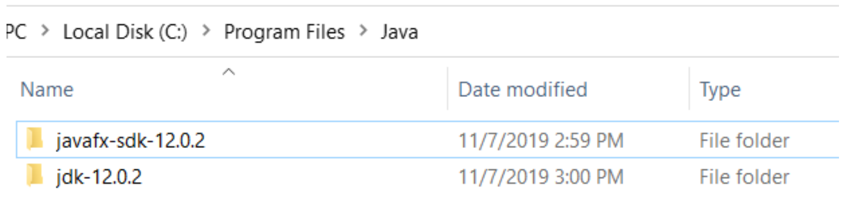
In this assignment, you need to have JavaFX in order to run the project. Don't worry, you don't need to do anything with them yet. We will learn more about JavaFX in the following lecture.

For now, if you already have any JavaFX version 11 to 15, you can use them for this assignment. **DO NOT USE version 16 or more since it may not draw our screen correctly**.

If you don't have any JavaFX, however, you can get the latest version from https://gluonhq.com/products/javafx/

Please download the latest release SDK of your OS version. Make sure you tick “include older versions” to make all versions available.

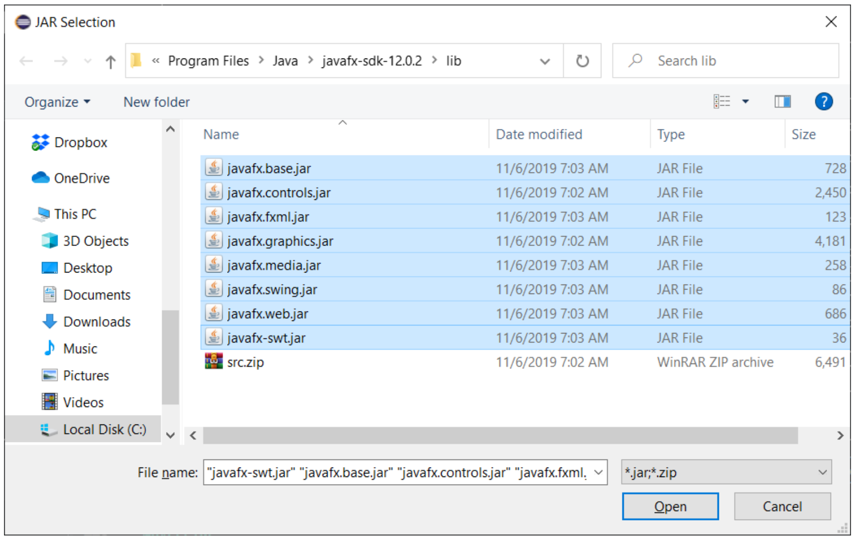
Once extracted the file, move it to somewhere that you think is easily accessible. The recommended location is “C:\Program Files\Java”


PC > Local Disk (C:) > Program Files > Java

Name	Date modified	Type
javafx-sdk-12.0.2	11/7/2019 2:59 PM	File folder
jdk-12.0.2	11/7/2019 3:00 PM	File folder

To add JavaFX to the project, follow this step.

1. Right-Click your project > Build Path… > Configure Build Path
2. In the Libraries Tab, under Classpath, click **Add External JAR…**
3. Navigate to the previously extracted JavaFX folder, go to the folder **lib**, and select every jar file in there and click Open.



Name	Date modified	Type	Size
javafx.base.jar	11/6/2019 7:03 AM	JAR File	728
javafx.controls.jar	11/6/2019 7:02 AM	JAR File	2,450
javafx.fxml.jar	11/6/2019 7:03 AM	JAR File	123
javafx.graphics.jar	11/6/2019 7:02 AM	JAR File	4,181
javafx.media.jar	11/6/2019 7:03 AM	JAR File	258
javafx.swing.jar	11/6/2019 7:03 AM	JAR File	86
javafx.web.jar	11/6/2019 7:03 AM	JAR File	686
javafx-swt.jar	11/6/2019 7:03 AM	JAR File	36
src.zip	11/6/2019 7:02 AM	WinRAR ZIP archive	6,491

File name: "javafx-swt.jar" "javafx.base.jar" "javafx.controls.jar" "javafx.fxml.

.jar;.zip

Almost there! You still need to do one last thing before we can run the program. You need to modify **Run Configurations**.
This can be done by right clicking Main.java > Run As > Run Configurations…
Under Arguments tab, in the VM arguments section, place this command in

```
--module-path "<path to your JavaFX folder>\lib" --add-modules
 javafx.controls,javafx.graphics,javafx.media,javafx.fxml
```

For example,

```
--module-path "C:\Program Files\Java\javafx-sdk-12.0.2\lib" --
add-modules javafx.controls,javafx.graphics,javafx.media,javafx.fxml
```

Afterwards, click Run and you will be able to run the program from now on.

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)

You will also need to modify some of the code in the Main.java under `application` package. The details will be specified later in the document.

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, Your own test cases are 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.
Anything else such as Rendering is already handled for you in this Exercise.

** 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 to complete this assignment. However, you might need to use these methods to help with your implementation.

2.1.1 Enum `Direction`

This enum represents direction. It contains the following values: LEFT, RIGHT, UP, DOWN and NONE.

Example Usage:

`Direction dir = Direction.LEFT;` //This represent the value of left direction.

2.1.2 Class `GameController`

This class is the game system. Most of the game's global variable are kept here.

2.1.2.1 Methods

+ void IntializeMap()	Load the data and initialize he sample map, as well as resets all the global variables.
+ GameMap getCurrentMap()	Get the active map
+ void movePlayer(Direction dir)	Move the player with the direction `dir`. (It just call method `move(Direction dir)` of the Player object) This also calls method `update()` on the Entity that implements the interface Updatable.
+ int getCoinCount() + void setCoinCount(int coin_count) + void addCoinCount(int coin_count)	Getter/Setter/Add for the field `coin_count`, which is the amount of the Coin player has collected.
+ boolean isGameWin() + void setGameWin(boolean is_win)	Getter/Setter for the field `is_win`, which is the status if the player has won or not.
+ boolean getGameSwitchStatus() + void setGameSwitchStatus(boolean game_switch)	Getter/Setter for the field `game_switch`, which is the status of the in-game Switch object.

2.1.3 Class `GameMap`

This class represent Map, which is a grid that contains many Cells.

2.1.3.1 Constructors

+ GameMap(int column,int row)	Intialize an empty map with the specified size.
+ GameMap(String[][] map)	Initialize a map with the content loaded from `String[][] map`

2.1.3.2 Methods

+ boolean addEntity(Entity e,int x, int y)	Adds Entity e into the cell at position (x,y) This will trigger Cell's `setEntity(Entity e)` method
+ Entity getEntity(int x,int y)	Get Entity of the cell at the position (x,y)
+ void removeEntity(int x,int y)	Removes Entity from the cell at the position (x,y)
+ isMovePossible(int targetx,int targety,Entity e)	Check if it is possible or not to move `Entity e` to the target position (targetx,targety) If the target position is empty, it returns true. Otherwise, return false unless the Entity at the target cell implements the interface Interactable, in which it triggers and check the result of the interaction using `interact(Entity e)` instead. It is guaranteed that this method will return false if the target position is outside the map.
+ ArrayList<Entity> getAllEntity()	Get the list of all Entity in the map

2.1.4 Class `Cell`

This class represents a cell, which is a single square in a map grid. Only one Entity can be in one cell at a time.

2.1.4.1 Methods

+ boolean IsEmpty()	Returns true if the cell is empty
+ boolean setEntity(Entity e)	Set the Entity e to this cell. If the cell is empty, then the Entity e will be assign as the Entity for this cell and returns true Otherwise, return false and trigger method `consume(Entity e)` if the Entity that occupied this cell implements the interface Consumable.
+ Entity getEntity()	Get the current Entity that occupied this cell.
+ void removeEntity()	Remove the current Entity from this cell.

2.1.5 Class `Sprites`

This class contains the constant using for rendering. It is only for organization purpose. Usage for this class will be specified later.

2.2 package `entity.base`

2.2.1 Abstract Class `Entity`

This class is the base class for all Entity in the map. This class ensures that every Entity has enough methods to work.

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

2.2.1.1 Constructor

+ Entity ()	Initialize the private fields.

2.2.1.2 Methods

+ *int getSymbol()*	This method returns the index symbol that

	represent the Entity during rendering. Do not worry about this one as the value for each type will be specified below.
+ boolean move(Direction dir)	Move the current Entity one unit in the specified direction. This also calls `isMovePossible (targetx, targety,this)` on the current game map to check if it can move or not. This method returns true if the move is successful, false if not. It is guaranteed that the result is always correct if you are implemented other stuff correctly.
+ void remove();	Remove this Entity from the board
+ int getDirection() + void setDirection(int direction)	Getter/Setter for the field `direction` Which is the direction that the Entity currently facing. It is always set each time it moves.

2.2.2 Interface `Interactable`

This interface defines methods for Entity that can be interacted with.

2.2.2.1 Method

+ *boolean interact(Entity e)*	This method is called when the `Entity e` checks for moveable space. It returns true if the results of the interaction let the `Entity e` pass through. Otherwise, return false. **Note:** Anything that occur during the moment of the interaction should be placed here as well. More detail will be provided in each Object.

2.2.3 Interface `Consumable`

This interface defines methods for Entity that consumes another Entity when it overlapped.

2.2.2.1 Method

+ boolean consumes(Entity e)	This method is called when the `Entity e` is being added onto the same space. It returns true if the main Entity can consume `Entity e`. Otherwise, return false. **Note:** Anything that occur during consuming should be placed here as well. More detail will be provided in each Object.

2.2.4 Interface `Updatable`

This interface defines methods for Entity that is can update itself once every player action.

2.2.2.1 Method

+ void update() throws IllegalValueException	This method is called when the player made a move. Do note that it is possible that this method can throws an exception. **Note:** Do not add or remove (including move) any Entity during this step. As it can cause a problem when iterating through Entity list. It is guaranteed that you will not do that during this assignment.
+ void valueCorrection()	This method should be called to reset the incorrect value when encountering an exception.

	You need to modify code inside Main.java so that this method is called when handling an exception from update() **Hint:** Use try…catch

2.3 package `entity`

This package contains implementation of the **concrete** Entity. Only the class that you need to create will be listed here. All classes must be created from scratch.

2.3.1 Class `Box`

Implements: Interactable

This class represent Box type Entity, which can be pushed around.

2.3.1.1 Methods

+ int getSymbol()	return the value Sprites.BOX Which contains the correct value of the subimage used to render the box.
+ boolean interact(Entity e)	Move the box in the same direction as `Entity e` Returns the result of the move

2.3.2 Class `Coin`

Implements: Interactable

This class represent Coin type Entity, which can be collected by either player or pushing things to it.

2.3.2.1 Methods

+ int getSymbol()	return the value Sprites.COIN Which contains the correct value of the subimage used to render the coin.
+ boolean interact(Entity e)	Remove the Coin and increment coin counts in GameController by 1. This method returns true because the coin is passable after it has been collected.

2.3.3 Class `Flag`

Implements: Interactable

This class represent Flag type Entity, which can be collected by player only. Player cannot push anything on top of it to collect, unlike Coins.

2.3.3.1 Methods

+ int getSymbol()	return the value Sprites.FLAG Which contains the correct value of the subimage used to render the flag.
+ boolean interact(Entity e)	If the `Entity e` is Player, then remove the Flag and set win status of the game in GameController. Then returns true afterward. For other type of `Entity e`, returns false. To make them not be able to pass through.

2.3.4 Class `Door`

Implements: Interactable, Consumable

This class represent Door type Entity, it blocks anything for the most part. Except if it collide with the key, it disappear.

2.3.4.1 Methods

+ int getSymbol()	return the value Sprites.DOOR Which contains the correct value of the subimage used to render the door.
+ boolean interact(Entity e)	Returns true if `Entity e` is Key. Otherwise returns false. This is because it needs to be overlapped with Key to trigger consume() method.
+ boolean consume(Entity e)	Since this Entity can only consumes Key, it returns true if `Entity e` is Key, and then remove itself. Otherwise returns false.

2.3.5 Class `Switch`

Implements: Interactable, Updatable

This class represent Switch type Entity. It is a solid object that flips switch every time the player interacts with it. If there are multiple switches on the screen, all switches should be updated to match the global Game Switch state.

2.3.5.1 Fields

- boolean isActive	A boolean to keep track if the switch has been flipped or not.

2.3.5.2 Constructor

+ Switch()	Initialize the `isActive` with the value from the GameController.

2.3.5.3 Methods

+ int getSymbol()	return the value Sprites.SWITCH_ON if the switch is active or Sprites.SWITCH_OFF otherwise. These contains the correct value of the subimage used to render the switch.
+ boolean interact(Entity e)	If the `Entity e` is Player, then flip the game switch state in GameController. This method always returns false. As this is solid object.
+ void update()	Update the switch's own `isActive` field value to match global game switch state in GameController.
+ void valueCorrection()	Since there is no Exception being thrown here, this method can be left empty.
+ boolean isActive() + void setActive(boolean isActive)	Getter/Setter for `isActive`

2.3.6 Class `ColorBox`

Implements: Interactable

This class represent Colored Box type Entity. It is a solid object that can be pushed around like Box, but only when the game's global switch matched their internal value.

2.3.6.1 Fields

- boolean activeBool	A boolean contains the game switch state that the box can be pushed.

2.3.6.2 Constructor

+ ColorBox (boolean activeBox)	Initialize `activeBool`

2.3.6.3 Methods

+ int getSymbol()	return the value Sprites.BOX_RED if the box active on the switch value `true` or Sprites.SWITCH_BLUE otherwise. These contains the correct value of the subimage used to render the colored box.
+ boolean interact(Entity e)	If the game's global switch matches the `activeBool`, move the box in the same direction as `Entity e` and returns the result of the move. Otherwise, returns false.
+ boolean getActiveBool() + void setActiveBool(boolean activeBool)	Getter/Setter for `activeBool`

2.3.7 Class `TrashCompactor`

Implements: Interactable, Consumable, Updatable

This class represent Trash Compactor type Entity. It is a solid object. However, it can consume any type of Box (Box, ColoredBox) if pushed into it, and then entering cooldown mode. During the cooldown mode, it cannot consume any Box. The cooldown timer ticks down once the player made a movement.

2.3.7.1 Fields

- int cooldown	A cooldown time, the Trash Compactor is only functional if the cooldown time is 0.

2.3.7.2 Constructor

+ TrashCompactor ()	Initialize `cooldown` with 0

2.3.7.3 Methods

+ int getSymbol()	return the value `Sprites.COMPACTOR_ON` if the Trash Compactor is usable. Returns `Sprites.COMPACTOR_OFF` otherwise These contains the correct value of the subimage used to render the Trash Compactor.
+ boolean interact(Entity e)	Returns true if the `Entity e` is a type of box and the Compactor is active to allow them to pass. Returns false otherwise to block them. **Tip:** You can use `Entity.isBox(Entity e)` to check if Entity e is a type of box or not.
+ boolean consume(Entity e)	Returns true if the `Entity e` is a type of box and the Compactor is usable to consume the Entity, as well as setting the cooldown time to the value of `GameController.MAX_COOLDOWN_TIME`. Returns false otherwise. **Tip:** You can use `Entity.isBox(Entity e)` to check if Entity e is a type of box or not.
+ void update() throws IllegalValueException	Decrease its `cooldown` value by 1 if it's more than 0 If `cooldown` is less than 0, throws IllegalValueException
+ void valueCorrection()	Set `cooldown` to 0
+ int getCooldown() + void setCooldown(int cooldown)	Getter/Setter for `cooldown`

Score Criteria

BoxTest		2.5
	testGetSymbol	0.5
	testInteract_SingleBox	1
	testInteract_MultiBox	1
CoinTest		2.5
	testGetSymbol	0.5
	testInteract_Player	1
	testInteract_Box	1
ColorBoxTest		2.5
	testGetSymbol	0.5
	testInteract_SingleBox_ON	0.5
	testInteract_SingleBox_OFF	0.5
	testInteract_MultiBox	1
DoorTest		4.5
	testGetSymbol	0.5
	testInteract_Key	1
	testInteract_Other	1
	testConsume_Key	1
	testConsume_Other	1
FlagTest		1.5
	testGetSymbol	0.5
	testInteract_Player	0.5
	testInteract_Box	0.5
SwitchTest		3.5
	testGetSymbol	0.5
	testInteract_Player	1
	testInteract_Other	1
	testUpdate	1
TrashCompactorTest		4.5
	testGetSymbol	0.5
	testInteract_Box_ON	0.5
	testInteract_Box_OFF	0.5
	testInteract_Other	0.5
	testConsume_Box_ON	0.5
	testConsume_Box_OFF	0.5
	testConsume_Other	0.5
	testUpdate	1
Exception		2
	Thrown in TrashCompactor	1
	Handled in Main	0.5
	Call valueCorrection	0.5
UML		2