German University in Cairo
Media Engineering and Technology
Prof. Dr. Slim Abdennadher
Dr. Nourhan Ehab
Dr. Ahmed Abdelfattah

CSEN 401 - Computer Programming Lab, *Spring 2024*
Attack on Titan: Utopia
Milestone 1
Deadline: 8.3.2024 @ 11:59 PM

This milestone is an *exercise* on the concepts of **Object Oriented Programming (OOP)**. The following sections describe the requirements of the milestone.

By the **end of this milestone**, you should have:

- A structured package hierarchy for your code.
- An initial implementation for all the needed data structures.
- Basic data loading functionality from a csv file.

1 Build the Project Hierarchy

1.1 Add the packages

Create a new `Java` project and build the following package hierarchy:

1. `game.engine`
2. `game.engine.base`
3. `game.engine.dataloader`
4. `game.engine.exceptions`
5. `game.engine.interfaces`
6. `game.engine.lanes`
7. `game.engine.titans`
8. `game.engine.weapons`
9. `game.engine.weapons.factory`
10. `game.tests`
11. `game.gui`

Afterwards, proceed by implementing the following classes. You are allowed to add more classes, attributes and methods. However, you must use the same specific names for the provided ones.

1.2 Naming and privacy conventions

Please note that all your class attributes must be `private` and all methods should be `public` unless otherwise stated. Your implementation should have the appropriate setters and getters conforming with the access constraints. if a variable is said to be READ then we are allowed to get its value. If the variable is said to be WRITE then we are allowed to change its value. Please note that getters and setters should match the Java naming conventions unless mentioned otherwise. If the instance variable is of type boolean, the getter method name starts by **is** followed by the **exact** name of the instance variable. Otherwise, the method name starts by the verb (get or set) followed by the **exact** name of the instance variable; the first letter of the instance variable should be capitalized. Please note that the method names are case sensitive.

Example 1 *You want a getter for an instance variable called `milkCount` → Method name = `getMilkCount()`*

Example 2 *You want a setter for an instance variable called `milkCount` → Method name = `setMilkCount()`*

Example 3 *You want a getter for a boolean variable called `alive` → Method name = `isAlive()`*

Throughout the whole milestone, if an attribute will never be changed once initialized, you should declare it as `final`. If a particular member (variable or method) belongs to the class, rather than instances of the class then it's said to be `static`. Combining both is commonly used for defining constants that should be accessible without creating an instance of the class and whose value remains constant throughout the program.

Example 4 *You want a constant price value for each instance → `final int price = 100;`*

Example 5 *You want a counter that will be shared by all instances → `static int counter = 0;`*

Example 6 *You want a constant code shared by all instances → `static final int code = 1;`*

1.3 Comparing Objects

In Java, the Comparable interface is used to impose a natural ordering on the objects of each class that implements it. The Comparable interface defines a single method, `compareTo()`, which is used to compare the current object with another object for order based on some criteria. To use comparable, a class must implement the `Comparable` interface and define the `compareTo()` method. This method takes the object to be compared with the current one and returns an integer that is equal to:

1. A negative integer → current object is less than the specified object.
2. Zero → current object is equal to the specified object.
3. A positive integer → current object is greater than the specified object.

Example 7 *Suppose you have a `Cat` class, and you want to sort cats by their weight*

```
public class Cat implements Comparable<Cat> {
    private String name;
    private int weight;

    public Cat(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public int compareTo(Cat cat) {
        return this.weight - cat.weight;
    }
```

2 Build the Interfaces

Interfaces define a set of abstract functionalities that a class can implement, allowing it to perform specific actions, like the comparing mentioned above. Classes should implement the interfaces that allow them to do their specified functionality.

2.1 Build the Attackee Interface

Name : `Attackee`

Package : `game.engine.interfaces`

Type : Interface

Description : Interface containing the methods available to all objects that gets attacked within the game.

2.1.1 Methods

1. `int getCurrentHealth()`: A getter method that retrieves the Attackee's current health.
2. `void setCurrentHealth(int health)`: A setter method that changes the Attackee's current health to the input value.
3. `int getResourcesValue()`: A getter method that retrieves the Attackee's resources value.

2.2 Build the Attacker Interface

Name : `Attacker`

Package : `game.engine.interfaces`

Type : Interface

Description : Interface containing the methods available to all attackers within the game.

2.2.1 Methods

1. `int getDamage()`: A getter method that retrieves the damage value that the attacker inflicts.

2.3 Build the Mobil Interface

Name : `Mobil`

Package : `game.engine.interfaces`

Type : Interface

Description : Interface containing the methods available to all objects that has mobility (i.e can move) within the game.

2.3.1 Methods

1. `int getDistance()`: A getter method that retrieves the Mobil's distance from its target.
2. `void setDistance(int distance)`: A setter method that changes the Mobil's distance from its target to the input value.
3. `int getSpeed()`: A getter method that retrieves the Mobil's movement speed.
4. `void setSpeed(int speed)`: A setter method that changes the Mobil's movement speed to the input value.

3 Build the Enums

3.1 Build the Battle Phase Enum

Name : `BattlePhase`

Package : `engine`

Type : Enum

Description : An enum representing the three phases of the game play. Possible values are: EARLY, INTENSE, GRUMBLING.

4 Build the Classes

You need to think about the boundary values of all variables in the setters. For example, you need to make sure that certain variables don't fall below zero.

4.1 Build the Titan Class

Name : `Titan`

Package : `game.engine.titans`

Type : Abstract class.

Description : A class representing the `Titan`s available in the game. A titan class can do the following functionalities:

1. move
2. attack
3. gets attacked
4. gets compared based on it's distance from the base/wall

No objects of type `Titan` can be instantiated.

4.1.1 Attributes

All the class attributes are of type `int` and READ ONLY unless stated otherwise.

1. `baseHealth`: An integer representing the original titan's health when spawned. This attribute will never be changed once initialized.
2. `currentHealth`: An integer representing the current titan's health which by default is equal to `baseHealth`. This attribute is READ and WRITE through a relevant interface's method.
3. `baseDamage`: An integer representing the amount of damage that a titan would cause when attacking a wall. This attribute will never be changed once initialized. This attribute is READ ONLY through a relevant interface's method.
4. `heightInMeters`: An integer representing the titan's height. This attribute will never be changed once initialized.
5. `distanceFromBase`: An integer representing the distance of a titan from the base (i.e wall). This attribute is READ and WRITE through a relevant interface's method.
6. `speed`: An integer representing the speed of a titan (i.e the distance a titan moves each turn). This attribute is READ and WRITE through a relevant interface's method.
7. `resourcesValue`: An integer representing the amount of resources that are gained by defeating the titan. This attribute will never be changed once initialized. This attribute is READ ONLY through a relevant interface's method.
8. `dangerLevel`: An integer representing the danger level of a titan. The smaller the value, the less dangerous that titan is. This attribute will never be changed once initialized.

4.1.2 Constructors

1. `Titan(int baseHealth, int baseDamage, int heightInMeters, int distanceFromBase, int speed, int resourcesValue, int dangerLevel)`: Constructor that initializes a `Titan` object with the given parameters as the attributes and initially sets the `currentHealth` with the `baseHealth`.

4.1.3 Methods

1. `int compareTo(Titan o)`: override the `Comparable` interface `compareTo` method to order titans based on their distance from the base/wall.

4.1.4 Subclasses

The game features four distinct types of titans, each represented as a subclass within the `Titan` class hierarchy. These subclasses should be organized in the same package where the `Titan` class is located. Every subclass, corresponding to a specific titan type, must define a unique constant `int TITAN_CODE`. This constant, consistent across all instances of a particular subclass, serves as an identifier and is assigned a number as per the provided table and can not be modified once initialized. Additionally, constructors in each subclass should be thoughtfully designed to extend the functionality of the `Titan` class's constructor, Carefully, initialize the setup for each titan type. The following list gives the different types of titans and their value of `TITAN_CODE`.

Titan's Types	TITAN_CODE
Pure Titan	1
Abnormal Titan	2
Armored Titan	3
Colossal Titan	4

4.2 Build the Pure Titan Class

Name : `PureTitan`

Package : `game.engine.titans`

Type : Class.

Description : A class representing the `PureTitan`s available in the game. This class is a subclass of the `Titan` class.

4.2.1 Constructors

1. `PureTitan(int baseHealth, int baseDamage, int heightInMeters, int distanceFromBase, int speed, int resourcesValue, int dangerLevel)`: Constructor that initializes a `PureTitan` object with the given parameters as the attributes by calling the super constructor.

4.3 Build the Abnormal Titan Class

Name : `AbnormalTitan`

Package : `game.engine.titans`

Type : Class.

Description : A class representing the `AbnormalTitan`s available in the game. This class is a subclass of the `Titan` class.

4.3.1 Constructors

1. `AbnormalTitan(int baseHealth, int baseDamage, int heightInMeters, int distanceFromBase, int speed, int resourcesValue, int dangerLevel)`: Constructor that initializes an `AbnormalTitan` object with the given parameters as the attributes by calling the super constructor.

4.4 Build the Armored Titan Class

Name : `ArmoredTitan`

Package : `game.engine.titans`

Type : Class.

Description : A class representing the `ArmoredTitan`s available in the game. This class is a subclass of the `Titan` class.

4.4.1 Constructors

1. `ArmoredTitan(int baseHealth, int baseDamage, int heightInMeters, int distanceFromBase, int speed, int resourcesValue, int dangerLevel)`: Constructor that initializes an `ArmoredTitan` object with the given parameters as the attributes by calling the super constructor.

4.5 Build the Colossal Titan Class

Name : `ColossalTitan`

Package : `game.engine.titans`

Type : Class.

Description : A class representing the `ColossalTitan`s available in the game. This class is a subclass of the `Titan` class.

4.5.1 Constructors

1. `ColossalTitan(int baseHealth, int baseDamage, int heightInMeters, int distanceFromBase, int speed, int resourcesValue, int dangerLevel)`: Constructor that initializes a `ColossalTitan` object with the given parameters as the attributes by calling the super constructor.

4.6 Build the Weapon Class

Name : `Weapon`

Package : `game.engine.weapons`

Type : Abstract class.

Description : A class representing the `Weapon`s available in the game. A weapon should be able to perform an attack.
No objects of type `Weapon` can be instantiated.

4.6.1 Attributes

1. `int baseDamage`: An integer representing the amount of damage a weapon can inflict. This attribute will not be changed once initialized. This attribute is READ ONLY through a relevant interface's method.

4.6.2 Constructors

1. `Weapon(int baseDamage)`: Constructor that initializes a `weapon` object with the given `baseDamage`.

4.6.3 Subclasses

The game features four distinct types of weapons, each represented as a subclass within the `Weapon` class hierarchy. These subclasses should be organized in the same package where the `Weapon` class is located. Every subclass, corresponding to a specific weapon type, must define a unique constant `int WEAPON_CODE`. This constant, consistent across all instances of a particular subclass, serves as an identifier and is assigned a number as per the below table and can not be modified once initialized. Additionally, constructors in each subclass should be thoughtfully designed to extend the functionality of the `Weapon` class's constructor, Carefully, initiaalize the setup for each weapon type. The following list gives the different types of weapons and their value of `WEAPON_CODE`.

Weapon's Types	WEAPON_CODE
Piercing Cannon	1
Sniper Cannon	2
VolleySpread Cannon	3
Wall Trap	4

4.7 Build the Piercing Cannon Class

Name : `PiercingCannon`

Package : `game.engine.Weapons`

Type : Class.

Description : A class representing the `PiercingCannon`s available in the game. This class is a subclass of the `Weapon` class.

4.7.1 Constructors

1. `PiercingCannon(int baseDamage)`: Constructor that initializes an `PiercingCannon` object with the `baseDamage` as the attribute by calling the super constructor.

4.8 Build the Sniper Cannon Class

Name : `SniperCannon`

Package : `game.engine.Weapons`

Type : Class.

Description : A class representing the `SniperCannon`s available in the game. This class is a subclass of the `Weapon` class.

4.8.1 Constructors

1. `SniperCannon(int baseDamage)`: Constructor that initializes an `SniperCannon` object with the `baseDamage` as the attribute by calling the super constructor.

4.9 Build the VolleySpread Cannon Class

Name : `VolleySpreadCannon`

Package : `game.engine.Weapons`

Type : Class.

Description : A class representing the `VolleySpreadCannon`s available in the game. This class is a subclass of the `Weapon` class.

4.9.1 Attributes

VolleySpread Cannon has two extra attributes other than `baseDamage`. These attributes are READ ONLY and does not change once initialized. Their main purpose is to specify a range of distance to inflict damage on titans (i.e. nothing happens to a titan outside this range)

1. `int minRange`: An integer representing the lower bound of the range. This attribute will never be changed once initialized.
2. `int maxRange`: An integer representing the upper bound of the range. This attribute will never be changed once initialized.

4.9.2 Constructors

1. `VolleySpreadCannon(int baseDamage, int minRange, int maxRange)`: Constructor that initializes an `VolleySpreadCannon` object with the `baseDamage` as the attribute by calling the super constructor and initializing the two extra attributes with the given parameters.

4.10 Build the Wall Trap Class

Name : `WallTrap`

Package : `game.engine.Weapons`

Type : Class.

Description : A class representing the `WallTrap`s available in the game. This class is a subclass of the `Weapon` class.

4.10.1 Constructors

1. `WallTrap(int baseDamage)`: Constructor that initializes an `WallTrap` object with the `baseDamage` as the attribute by calling the super constructor.

4.11 Build the Wall Class

Name : `Wall`

Package : `game.engine.base`

Type : Class.

Description : A class representing the `Wall`s in which a titan attacks in the game. A wall is a class that gets attacked. Its resources value is -1 and is not deducted from the player's resources if destroyed.

4.11.1 Attributes

1. `int baseHealth`: An integer representing the original value of the wall's health. This attribute is READ ONLY and gets initialized once.
2. `int currentHealth`: An integer representing the current titan's health which originally equals the `baseHealth`. This attribute is READ and WRITE through a relevant interface's method.

4.11.2 Constructors

1. `Wall(int baseHealth)`: Constructor that initializes a `Wall` object with the given `baseHealth`. `currentHealth` also starts with the `baseHealth`.

4.12 Build the Lane Class

Name : `Lane`

Package : `game.engine.lanes`

Type : Class.

Description : A class representing the `Lane`s in which a titan walk on to the wall. the class is able to get compared based on it's danger level.

4.12.1 Attributes

All attributes are READ ONLY.

1. `Wall laneWall`: A wall object found in the lane. This attribute will never be changed once initialized.
2. `int dangerLevel`: An integer representing the danger level of a lane based on the number and danger level of the titans on it. This attribute is READ and WRITE and is initially set to 0.
3. `PriorityQueue<Titan> titans`: A queue that stores all the titans in a given lane and is initially empty. Titans closer to the walls have a higher priority. This attribute will never be changed once initialized.
4. `ArrayList<Weapon> weapons`: A queue that stores all the weapons in a given lane and is initially empty. This attribute will never be changed once initialized.

4.12.2 Constructors

1. `Lane(Wall laneWall)`: Constructor that initializes a `Lane` object with the given `laneWall` and the rest of the attributes as the initial empty values.

4.12.3 Methods

1. `int compareTo(Lane o)`: override the `Comparable` interface `compareTo` method to order lanes based on their danger level.

4.13 Build the Titan Registry Class

Name : `TitanRegistry`

Package : `game.engine.titans`

Type : Class.

Description : A class representing the `TitanRegistry`, which is a place to store the titan's information that was read from the csv file for later use.

4.13.1 Attributes

All the class attributes are of type `int` and READ ONLY.

1. `code`: An integer representing the type of titan. This attribute will never be changed once initialized.
2. `baseHealth`
3. `baseDamage`
4. `heightInMeters`
5. `speed`
6. `resourcesValue`
7. `dangerLevel`

4.13.2 Constructors

1. `TitanRegistry(int code, int baseHealth, int baseDamage, int heightInMeters, int speed, int resourcesValue, int dangerLevel)`: Constructor that initializes a `TitanRegistry` object with the given parameters as the attributes.

4.14 Build the Weapon Registry Class

Name : `WeaponRegistry`

Package : `game.engine.weapons`

Type : Class.

Description : A class representing the `WeaponRegistry`, which is a place to store the weapon's information that was read from the csv file for later use.

4.14.1 Attributes

All the class attributes are of type `int` and READ ONLY unless stated otherwise.

1. `code`: An integer representing the type of weapon. This attribute will never be changed once initialized.
2. `price`: An integer representing the price of the weapon.
3. `damage`: Same as in `Weapon` class.
4. `String name`: A variable representing the weapon's name.
5. `minRange`: An integer representing the lower bound of a weapon's damage range.
6. `maxRange`: An integer representing the upper bound of a weapon's damage range.

4.14.2 Constructors

1. `WeaponRegistry(int code, int price)`: Constructors that initializes a `Weapon` object with only the code and price.
2. `WeaponRegistry(int code, int price, int damage, String name)`: Constructors that initializes a `Weapon` object with the given parameters as the attributes.
3. `WeaponRegistry(int code, int price, int damage, String name, int minRange, int maxRange)`: Constructors that initializes a `Weapon` object with the given parameters as the attributes.

4.15 Build the Data Loader Class

Name : `DataLoader`

Package : `game.engine.dataloader`

Type : Class.

Description : A class that is used to import data from a given csv file. The imported data is then used to create and initialize registries for Titans and Weapons and fill them with the entries from the csv files. The read file should be placed in the same directory as the .src folder and not inside it.

4.15.1 Attributes

All attributes are READ ONLY and initialized once unless otherwise specified. They also should be accessed at class level.

1. `String TITANS_FILE_NAME`: A String containing the name of the titan's csv file.
2. `String WEAPONS_FILE_NAME`: A String containing the name of the weapon's csv file.

4.16 Description of CSV files format

Titans

1. The titans are found in a file titled `titans.csv`.
2. Each line represents the information of a single titan.
3. The data has no header, i.e. the first line represents the first titan.
4. The parameters are separated by a comma (,).
5. The line represents the titans' data as follows: **code, baseHealth, baseDamage, heightInMeters, speed, resourcesValue, dangerLevel**.

Weapons

1. The weapons are found in a file titled `weapons.csv`.
2. Each line represents the information of a single weapon.
3. The data has no header, i.e. the first line represents the first weapon.
4. The parameters are separated by a comma (,).
5. The line represents the weapons' data as follows: **code, price, damage, name** or **code, price, damage, name, minRange, maxRange** based on the type of weapon (**minRange** and **maxRange** are added only if the weapon type is `VolleySpreadCannon`).

4.16.1 Methods

1. `public static HashMap<Integer, TitanRegistry> readTitanRegistry() throws IOException`: Reads the TITANS_FILE_NAME CSV file using `BufferedReader` and creates a TitanRegistry with the values, then loads it into the hashmap alongside it's code.
2. `public static HashMap<Integer, WeaponRegistry> readWeaponRegistry() throws IOException`: Reads the WEAPONS_FILE_NAME CSV file using `BufferedReader` and creates a WeaponRegistry with the values, then loads it into the hashmap alongside it's code.

4.17 Build the Factory Response Class

Name : `FactoryResponse`

Package : `game.engine.weapons.factory`

Type : Class.

Description : A class representing the `FactoryResponse`, which is an object to store the weapon that was bought with the remaining resources.

4.17.1 Attributes

All the class attributes are READ ONLY unless otherwise specified. Also, they will not be changed once initialized.

1. `Weapon weapon`: A weapon that was bought.
2. `int remainingResources`: An integer representing the remaining resources after buying the weapon.

4.17.2 Constructors

1. `FactoryResponse(Weapon weapon, int remainingResources)`: Constructors that initializes a `FactoryResponse` object with the given parameters as the attributes.

4.18 Build the Weapon Factory Class

Name : `WeaponFactory`

Package : `game.engine.weapons.factory`

Type : Class.

Description : A class representing the `WeaponFactory`, which is used to store the available weapons that can be purchased during each turn.

4.18.1 Attributes

1. `HashMap<Integer, WeaponRegistry> weaponShop`: A hashmap that keeps track of a `WeaponRegistry` that's read from the csv file alongside it's code. This attribute is initialized once the data is read and is READ ONLY.

4.18.2 Constructors

1. `WeaponFactory() throws IOException`: Constructors that initializes a `WeaponFactory` object where the hashmap is initialized by the data read from the csv file.

Game Setup

4.19 Build the Battle Class

Name : `Battle`

Package : `engine`

Type : Class

Description : A class representing the **Game itself. This class will represent the main engine that manages the flow of the game.**

4.19.1 Attributes

All the class attributes are READ ONLY unless stated otherwise.

1. `int[][] PHASES_APPROACHING_TITANS`: A 2D array containing titans' codes, representing the order of titans during each phase. This attribute is a constant at class level that's initialized only once.
2. `int WALL_BASE_HEALTH`: An integer representing the original health of any walls created throughout the game. This attribute will be set to 10000 and is a constant at class level that's initialized only once.
3. `int numberOfTurns`: An integer representing the number of turns during game play. This attribute is both READ and WRITE.
4. `int resourcesGathered`: An integer representing number of available resources throughout the game. Initially equal to the value of initial resources per lane * initial number of lanes. This attribute is both READ and WRITE.
5. `BattlePhase battlePhase`: An enum representing the current phase. This attribute is both READ and WRITE and initially starts as EARLY.
6. `int numberOfTitansPerTurn`: An integer representing number of titans in a turn throughout the game. This attribute is both READ and WRITE and initially starts with 1.
7. `int score`: An integer representing accumulated score throughout the game. This attribute is both READ and WRITE.

8. int titanSpawnDistance: An integer representing the distance a titan would be spawned at. This attribute is both READ and WRITE.
9. WeaponFactory weaponFactory: A WeaponFactory object to store the available weapons to be bought. This attribute will never be changed once initialized.
10. HashMap<Integer, TitanRegistry> titansArchives: A Hashmap containing an archive of titans based on their codes. Initially has the data read from the dataloader. This attribute will never be changed once initialized.
11. ArrayList<Titan> approachingTitans: An arraylist representing the coming titans during a turn. Will be treated as a queue (First in First out). This attribute will never be changed once initialized.
12. PriorityQueue<Lane> lanes: A queue containing the **active lanes based on their danger level. Least dangerous lanes will have the highest priority. This attribute will never be changed once initialized.**
13. **ArrayList<Lane> originalLanes: An arraylist containing all the lanes from the start of the game. This attribute will never be changed once initialized.**

4.19.2 Constructors

1. Battle(int numberOfTurns, int score, int titanSpawnDistance, int initialNumOfLanes, int initialResourcesPerLane) throws IOException: Constructor that generates the main battle and initializes it with the parameters and the rest of attributes mentioned above.

4.19.3 Methods

1. private void initializeLanes(int numOfLanes): The initializeLanes method creates and adds a specified number of lanes to the game. Each lane is initialized with its own wall having the predefined WALL_BASE_HEALTH. These lanes are then stored in two different collections within the game, the originalLanes and lanes.

5 Build the Exceptions

5.1 Build the Game Action Exception Class

Name : GameActionException

Package : game.engine.exceptions

Type : Abstract class.

Description : Class representing a generic exception that can occur during game play. These exceptions arise from any invalid action that is performed. This class is a subclass of java's Exception class. No objects of type GameActionException can be instantiated.

5.1.1 Constructors

1. GameActionException(): Initializes an instance of a GameActionException by calling the constructor of the super class.
2. GameActionException(String message): Initializes an instance of a GameActionException by calling the constructor of the super class with a customized message.

5.1.2 Subclasses

This class has two subclasses: InvalidLaneException and InsufficientResourcesException.

5.2 Build the Invalid Lane Exception Class

Name : `InvalidLaneException`

Package : `game.engine.exceptions`

Type : Class

Description : Class representing an exception that can occur whenever the player tries to buy a weapon in a destroyed or non existent lane. This class is a subclass of `GameActionException` class.

5.2.1 Attributes

1. `static final String MSG`: A string representing the message of the exception that should be initialized to "Action done on an invalid lane".

5.2.2 Constructors

1. `InvalidLaneException()`: Initializes an instance of a `InvalidLaneException` by calling the constructor of the super class with `MSG`.
2. `InvalidLaneException(String message)`: Initializes an instance of a `InvalidLaneException` by calling the constructor of the super class with a customized message.

5.3 Build the Insufficient Resources Exception Class

Name : `InsufficientResourcesException`

Package : `game.engine.exceptions`

Type : Class

Description : Class representing an exception that can occur whenever the player tries to buy a weapon without having enough resources. This class is a subclass of `GameActionException` class.

5.3.1 Attributes

1. `static final String MSG`: A string representing the message of the exception that should be initialized to "Not enough resources, resources provided = ".
2. `int resourcesProvided`: An integer representing the current available resources. This attribute is both READ and WRITE.

5.3.2 Constructors

1. `InsufficientResourcesException(int resourcesProvided)`: Initializes an instance of an `InsufficientResourcesException` by calling the constructor of the super class with a message including `MSG` as well as the `resourcesProvided` and setting the `resourcesProvided` with the given parameter.
2. `InsufficientResourcesException(String message, int resourcesProvided)`: Initializes an instance of a `InsufficientResourcesException` by calling the constructor of the super class with a customized message and setting the `resourcesProvided` with the given parameter.

5.4 Build the Invalid CSV Format Class

Name : `InvalidCSVFormat`

Package : `game.engine.exceptions`

Type : Class

Description : Class representing an exception that can occur whenever an invalid csv is being read. This class is a subclass of java's `IOException` class.

5.4.1 Attributes

1. `static final String MSG`: A string representing the message of the exception that should be initialized to ”Invalid input detected while reading csv file, input = \n”.
2. `String inputLine`: A variable representing the csv file name. This attribute is both READ and WRITE.

5.4.2 Constructors

1. `InvalidCSVFormat(String inputLine)`: Initializes an instance of a `InvalidCSVFormat` by calling the constructor of the super class with a message including `MSG` as well as the `inputLine` and setting the `inputLine` with the given parameter.
2. `InvalidCSVFormat(String message, String inputLine)`: Initializes an instance of a `InvalidCSVFormat` by calling the constructor of the super class with a customized message and setting the `inputLine` with the given parameter.