German University in Cairo
Media Engineering and Technology
Prof. Dr. Slim Abdennadher

Computer Programming Lab, *Spring 2022*
Marvel - Ultimate War
Milestone 1

Deadline: 31.3.2022 @ 23:59

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 packaging hierarchy for your code.
- An initial implementation for all the needed data structures.
- Basic data loading capabilities from a csv file.

1 Build the Project Hierarchy

1.1 Add the packages

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

1. `engine`
2. `exceptions`
3. `model.abilities`
4. `model.effects`
5. `model.world`
6. `tests`
7. `views`

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

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. You should implement the appropriate setters and getters conforming with the access constraints. Throughout the whole milestone, 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. 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()`

2 Build the (EffectType) Enum

Name : `EffectType`

Package : `model.effects`

Type : Enum

Description : An enum representing the different possible types of effects. Possible values are: BUFF and DEBUFF.

3 Build the (Effect) Class

Name : `Effect`

Package : `model.effects`

Type : Class

Description : A class representing `Effect`s in the game.

3.1 Attributes

All the class attributes are READ ONLY unless otherwise specified.

1. `String name`: A string representing the effect name.
2. `int duration`: An int representing the number of turns the effect remains on the target champion. This attribute is READ and WRITE.
3. `EffectType type`: The enum representing whether the effect is a BUFF (positive effect), or DEBUFF (negative effect).

3.2 Constructors

1. `Effect(String name, int duration, EffectType type)`: Constructor that initializes an `Effect` object with the given parameters as the attributes.

3.3 Subclasses

There are 10 different types of effects available in the game. Each Effect type is modelled as a subclass of the `Effect` class. Each effect type should be implemented in a separate class within the same package as the `Effect` class. Each of the subclasses representing the different effects should have its own constructor that utilizes the `Effect` constructor. Carefully consider the design of the constructor of each subclass.

The following list gives the different class names.

Class name	Type	Class name	Type
`Disarm`	*DEBUFF*	`Embrace`	*BUFF*
`PowerUp`	*BUFF*	`Root`	*DEBUFF*
`Shield`	*BUFF*	`Shock`	*DEBUFF*
`Silence`	*DEBUFF*	`Dodge`	*BUFF*
`SpeedUp`	*BUFF*	`Stun`	*DEBUFF*

4 Build the (AreaOfEffect) Enum

Name : `AreaOfEffect`

Package : `model.abilities`

Type : Enum

Description : An enum representing the different possible areas of effects for `Ability`s. Possible values are: SELFTARGET, SINGLETARGET, TEAMTARGET, DIRECTIONAL, and SURROUND.

5 Build the (Ability) Class

Name : `Ability`

Package : `model.abilities`

Type : Class

Description : A class representing `Abilitiess` in the game.

5.1 Attributes

All the class attributes are READ ONLY unless otherwise specified.

1. `String name`: A string representing the name of the ability.
2. `int manaCost`: An int representing the mana cost of the ability.
3. `int baseCooldown`: An int representing the number of turns a champion must wait after casting the ability in order to cast it again.
4. `int currentCooldown`: An int representing the number of turns that the champion is currently waiting for in order to recast the ability. This attribute is READ and WRITE.
5. `int castRange`: An int representing the maximum cast range of the ability.
6. `int requiredActionPoints`: An int representing the needed action points to cast the ability.
7. `AreaOfEffect castArea`: The enum representing the area of effect of the ability.

5.2 Constructors

1. `Ability(String name,int cost, int baseCoolDown, int castRange, AreaOfEffect area , int required)`: Constructor that initializes an `Ability` object with the given parameters as the attributes.

5.3 Subclasses

There are 3 different types of abilities available in the game. Each ability type is modelled as a subclass of the `Ability` class. Each ability type should be implemented in a separate class within the same package as the `Ability` class. Each of the subclasses representing the different ability types should have its own constructor that utilizes the `Ability` constructor, and take any extra attributes as parameters. Carefully consider the design of the constructor of each subclass.

The following list gives the different class names.

Class name	Extra attributes	access
`DamagingAbility`	int damageAmount	READ and WRITE
`HealingAbility`	int healAmount	READ and WRITE
`CrowdControlAbility`	Effect effect	READ ONLY

World Setup

6 Build the (Direction) Enum

Name : `Direction`

Package : `model.world`

Type : Enum

Description : An enum representing the different possible moving/ casting directions. Possible values are: RIGHT, LEFT, UP, and DOWN.

7 Build the (Condition) Enum

Name : `Condition`

Package : `model.world`

Type : Enum

Description : An enum representing the different possible conditions a champion may be in. Possible values are: ACTIVE, INACTIVE, ROOTED and KNOCKEDOUT.

8 Build the (Cover) Class

Name : `Cover`

Package : `model.world`

Type : Class

Description : A class representing `Cover`s in the game.

8.1 Attributes

All the class attributes are READ and WRITE unless otherwise specified.

1. `int currentHP`: The `Cover`'s current Health points. This value must never be below zero.
2. `Point location`: holds the `Cover`'s location on the board. This attribute is READ ONLY. The `Point` here is the Java built-in Point in the `AWT` package.

8.2 Constructors

1. `Cover(int x, int y)`: Constructor that initializes a `Cover` object with the given (x,y) as the `Cover`'s location. It should also set the `Cover`'s health points with a random number between 100 (inclusive) and 1000 (exclusive).

9 Build the (Champion) Class

Name : `Champion`

Package : `model.world`

Type : Class

Description : A class representing `Champion`s in the game.

9.1 Attributes

All the class attributes are READ and WRITE unless otherwise specified.

1. `String name`: The champion's name. This attribute is READ ONLY.
2. `int maxHP`: The maximum health points belonging to this champion. This is the upper bound of champion's `currentHP`. This attribute is READ ONLY
3. `int currentHP`: An integer representing the current health points for the champion.
4. `int mana`: An integer representing the amount of mana available for the champion throughout the entire game.This attribute is READ and WRITE.
5. `int maxActionPointsPerTurn`: An integer representing the number of action points the champions receives every turn.
6. `int currentActionPoints`: An integer representing the amount of action points remaining for the champion in their current turn. This attribute is READ and WRITE.
7. `int attackRange`: An integer representing the normal attack range for the champion. This attribute is READ ONLY.
8. `int attackDamage`: This number represents the damage inflicted on champions or cover when the champion performs a normal attack.
9. `int speed`: This number represents the speed attribute for the champion, and will influence when the champion's turn takes place.
10. `ArrayList<Ability> abilities`: An arraylist containing all the available abilities for the champion. This attribute is READ ONLY.
11. `ArrayList<Effect> appliedEffects`: An arraylist containing all the current effects being applied on the champion. This attribute is READ ONLY.
12. `Condition condition`: An attribute representing the champion's state in the game. Any champion starts as ACTIVE.
13. `Point location`: A point representing the champion's location on the board. The `Point` here is the Java built-in Point in the `AWT` package.

9.2 Constructors

1. `Champion(String name, int maxHP, int mana, int maxActions, int speed, int attackRange, int attackDamage)`:

9.3 Subclasses

There are 3 different types of champions available in the game. Each champion type is modelled as a subclass of the `Champion` class. Each champion type should be implemented in a separate class within the same package as the `Champion` class. Each of the subclasses representing the different champion types should have its own constructor that utilizes the `Champion` constructor. Carefully consider the design of the constructor of each subclass.

The following list gives the different class names.

Class name
`AntiHero`
`Hero`
`Villain`

Game Setup

10 Build the (Player) Class

Name : `Player`

Package : `engine`

Type : Class

Description : A class representing a `Player` in the game.

10.1 Attributes

All the class attributes are READ ONLY unless otherwise specified.

1. `String name`: The `Player`'s name.
2. `Champion leader`: The selected leader for the player. This attribute is READ and WRITE.
3. `ArrayList<Champion> team`: Contains the champions of the player's team .

10.2 Constructors

1. `Player(String name)`: Constructor that initializes a `Player` object with the given `name`.

11 Include the (PriorityQueue) class

Name : `PriorityQueue`

Package : `engine`

Type : Class

Description : An implementation of a PriorityQueue that will be used to establish the turn order in the game. Download the PriorityQueue.java file from the CMS and place it inside the `engine` package.

12 Build the (Game) Class

Name : `Game`

Package : `engine`

Type : Class

Description : A class representing the `Game` itself. This class will represent the main engine of the game, and will ensure all game rules are followed.

12.1 Attributes

All the class attributes are READ ONLY unless otherwise specified.

1. `Player firstPlayer`: The first player in the game.
2. `Player secondPlayer`: The second player in the game.
3. `boolean firstLeaderAbilityUsed`: A value representing whether the first player has used his leader ability.
4. `boolean secondLeaderAbilityUsed`: A value representing whether the second player has used his leader ability.

5. `Object[][] board`: The 2D grid representing the game arena.It is a 5x5 grid.
6. `static ArrayList<Champion> availableChampions`: Contains all champions available for both players to pick from.
7. `static ArrayList<Ability> availableAbilities`:Contains the different abilities of all available champions.
8. `PriorityQueue turnOrder`: Contains all characters available to pick from.
9. `static int BOARDHEIGHT`: A value representing the height of the board. This value can not be changed once initialized.
10. `static int BOARDWIDTH`: A value representing the width of the board.This value can not be changed once initialized.
    The below figure demonstrates the orientation of the board

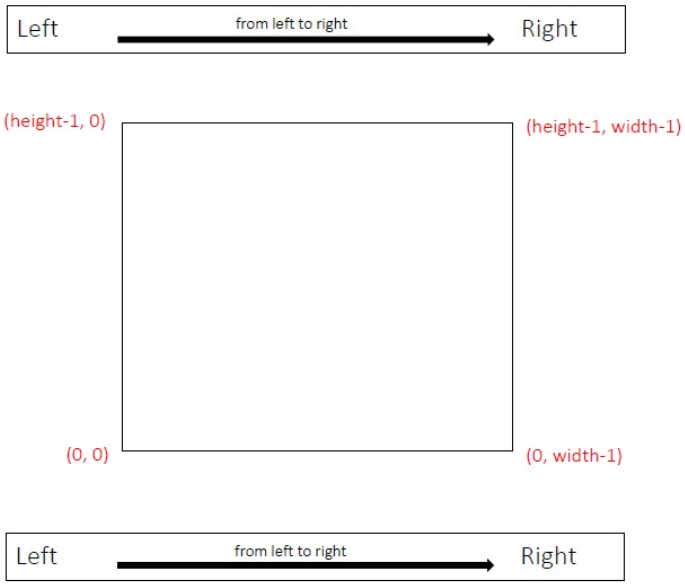

Figure 1: Board Orientation

12.2 Constructors

1. `Game(Player first, Player second)`: Constructor that initializes a `Game` with both players. It also places both player's champions on the board and distributes the covers.

12.3 Methods

1. `private void placeChampions()`: Places the champions of both players on the board. The first player's champions should be placed on the bottom edge adjacent to each other. The second player's champions should be placed on the upper edge adjacent to each other. No champion should be placed on any corner of the four corners of the board. For each player, the champions should be placed on their respective edge from left to right based on the order on which the player has chosen his champions with.
2. `private void placeCovers()`: Places five covers on five random empty cells on the board. No cover should be placed on any of the four corners of the board.
3. `public static void loadAbilities(String filePath)`: Reads the CSV file with `filePath` and loads the abilities into the `availableAbilities` ArrayList.

4. `public static void loadChampions(String filePath)`: Reads the CSV file with `filePath` and loads the champions into the `availableChampions` ArrayList.

12.4 Description of CSV files format

1. You should add `throws Exception` to the header of any constructor or method that reads from a csv file to compensate for any exceptions that could arise.
2. **Abilities**
   (a) The abilities are found in a file titled `Abilities.csv`.
   (b) Each line represents the information of a single ability.
   (c) The data has no header, i.e. the first line represents the first ability.
   (d) The parameters are separated by a comma (,).
   (e) The line represents the ability's data as follows: **Type, name, manaCost, castRange, baseCooldown, AreaOfEffect, requiredActionsPerTurn, damageAmount/healAmount/ effect name, effect duration (in case the ability is a CrowdControl ability)**.
   (f) In case of CrowdControl abilities, you must create an instance of the corresponding effect.
   (g) The type represents the type of ability:-
   - CC for CrowdControlAbility
   - DMG for DamagingAbility
   - HEL for HealingAbility

   Example 2 *An entry for an **Ability** will look like:*
   CC,Phase Shift,35,2,1,SINGLETARGET,3,Shield,2
3. **Champions**
   (a) The champions are found in a file titled `Champions.csv`.
   (b) Each line represents a champion.
   (c) The data has no header, i.e. the first line represents the first champion.
   (d) The parameters are separated by a comma (,).
   (e) The line represents the champion's data as follows: **Type, name, maxHP, mana, actions, speed, attackRange, attackDamage, ability1 name, ability2 name, ability3 name**.
   (f) The type represents the type of champion:-
   - 'A' for AntiHero
   - 'H' for Hero
   - 'V' for Villain

   Example 3 *An entry for a **Champion** will look like:*
   H,Captin America,1000,750,4,5,2,50,shield throw,I can do this all day,Shield Up

12.5 Hints on How to Read data from CSV file

- In order to read data from a file, you will need to use the pre-defined `BufferedReader` class which uses another pre-defined `FileReader` class.
- You need to place the csv files(s) that you want to read data from **beside** the **src** package of your project (not inside it). You can do that in eclipse by dragging and dropping the file (or copying and pasting it) on the **project folder** not the **src** package inside it.
- To initialize a `BufferedReader` Object that reades from a file called "test.csv", you can use the following piece of code: `BufferedReader br= new BufferedReader(new FileReader("test.csv");`

- You can get the content of the file line by line .By using the method `readLine()` that is pre-defined for the `BufferedReader` class, you can get the content of a single line and the line after it will be ready to be fetched once you call the method again with the same `BufferedReader`. The method will return the content of the line as a String or null if you reached the end of the file.
- Since each line of the csv file has its values separated by commas, you can use the method `split(",")` that is pre-defined for the `String` class to get the values around the commas.

Exceptions

13 Build the (GameActionException) Class

Name : `GameActionException`

Package : `exceptions`

Type : Class

Description : Class representing a generic exception that can occur during the game play. These exceptions arise from any invalid action that is performed. This class is a subclass of the java Exception class. This class has four subclasses; NotEnoughResourcesException, AbilityUseException, LeaderAbilityAlreadyUsedException, and UnallowedMovementException.

13.1 Constructors

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

14 Build the (NotEnoughResourcesException) Class

Name : `NotEnoughResourcesException`

Package : `exceptions`

Type : Class

Description : A subclass of `GameActionException` representing an exception that occurs upon trying to perform an action in the wrong turn.

14.1 Constructors

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

15 Build the (AbilityUseException) Class

Name : `AbilityUseException`

Package : `exceptions`

Type : Class

Description : A subclass of `GameActionException` representing an exception that occurs upon trying to cast an invalid ability.

15.1 Constructors

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

16 Build the (LeaderAbilityAlreadyUsedException) Class

Name : `LeaderAbilityAlreadyUsedException`

Package : `exceptions`

Type : Class

Description : A subclass of `GameActionException` representing an exception that occurs upon trying to reuse a leader ability, which is only allowed once.

16.1 Constructors

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

17 Build the (UnallowedMovementException) Class

Name : `UnallowedMovementException`

Package : `exceptions`

Type : Class

Description : A subclass of `GameActionException` representing an exception that occurs upon trying to preform an unallowed movement.

17.1 Constructors

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