170021928

CS5001 A2 Report

ANIMAL ADVANCE!

```
 _Y>_   Slingshot
 _H\_   Catapult
 _0o_   Cannon

  ~>    Rat
  MP    Camel
 /M@\   Elephant

|^|__|^| Castle
```

How to Run

I have delivered the game as a runnable JAR file. Run the game from the terminal thus:

java -jar towerDefence.jar <corridor length>

<corridor length> is an optional argument which you may use to specify the length of the game corridor (up to length 60). If it is not passed, the default length of 15 will be used.

Extensions

I have implemented all of the extensions:

- The player receives coins upon killing enemies which can be used to purchase new towers at any time.

- I have added one new type of enemy (the Camel) and one new type of tower (the Cannon).

- The user can play the game through a simple terminal-based interface, and see the resulting action play out on a map.

Report

Here I've walked through the game, explaining my design decisions along the way. My code is well commented, so please see the javadoc and the code itself for the fine detail of my implementation.

The Game class contains the main method and is where all the action happens. It contains the methods that drive the control of the game, and several final variables for things that don't change - i.e. the maximum possible corridor length, the purse starting value etc. When the

game is run, a new Game instance is created and a blank map is initialised.

I created a Map class to keep this and other related map methods in one place, such as checking if a space on the map is free, and printing the map given the array lists of enemies and towers. The Map class also has some private variables which are used to create the map and do not change, such as the height and the position of the castle. The top two rows are reserved for the tower position numbers (which just make it easier for the user to know where they want to place their towers) and below that, the towers themselves. The Map constructor requires one parameter, an int corridorLength which may be supplied by the user when the game is run, or may be a default value.

The user is provided with a purse containing a number of coins which may be used to buy towers. The number of coins is calculated relative to the length of the corridor : the longer the corridor, the easier the game and so fewer coins are provided. I considered creating a separate Purse class with setters and getters, but it seemed like overkill - all that is needed is a variable to hold the number of coins the user has, so I have implemented it as an attribute of the Game class.

The default starting purse contains 70 coins, but the number of coins actually provided to the player is calculated by subtracting the length of the corridor from this default. This is why the maximum corridor length is 60, as any higher number would leave the player with less than 10 coins at the start of the game, which is not enough to buy any towers at all.

Here you can see a newly initiated game with the default corridor length of 15. The castle on the right is what the user must defend, and the numbers are possible tower positions. The enemies will move from left to right across the map. The user may choose to spend some of their coins to purchase towers of varying kinds before the animals arrive. (Or they may choose to wait and see which animals appear first - this can be a good tactic! If there are lots of rats, you're better off with slingshots.)

```
8afbc818:CS5001-p2-oop JMCooper$ java -jar towerDefence.jar
  0    1    2    3    4    5    6    7    8    9    10   11   12   13   14  |^|__|^|
                                                                             |^|__|^|
                                                                             |^|__|^|
                                                                             |^|__|^|
                                                                             |^|__|^|
                                                                             |^|__|^|
                                                                             |^|__|^|
PURSE: 55
Enter 1 to buy a Slingshot: DMG: 1 DELAY: 1 COST: 10
Enter 2 to buy a Catapult:  DMG: 5 DELAY: 3 COST: 20
Enter 3 to buy a Cannon:    DMG: 10 DELAY: 5 COST: 30
Enter 4 to continue.
```

At the start of the game, a scanner is created to get user input from the console, and remains open until the game terminates. The user input is checked to make sure it is valid, and if they have chosen to create a tower one is instantiated with the provided position and added to the array list, and the map and purse are updated.

```
PURSE: 55
Enter 1 to buy a Slingshot: DMG: 1 DELAY: 1 COST: 10
Enter 2 to buy a Catapult:  DMG: 5 DELAY: 3 COST: 20
Enter 3 to buy a Cannon:    DMG: 10 DELAY: 5 COST: 30
Enter 4 to continue.
5
Sorry, that's not an option.
PURSE: 55
Enter 1 to buy a Slingshot: DMG: 1 DELAY: 1 COST: 10
Enter 2 to buy a Catapult:  DMG: 5 DELAY: 3 COST: 20
Enter 3 to buy a Cannon:    DMG: 10 DELAY: 5 COST: 30
Enter 4 to continue.
r
Give me a number, please.
1
Choose tower position from 0 to 14
13

  0    1    2    3    4    5    6    7    8    9    10   11   12   13   14  |^|__|^|
                                                        _Y>_       |^|__|^|
                                                                   |^|__|^|
                                                                   |^|__|^|
                                                                   |^|__|^|
                                                                   |^|__|^|
                                                                   |^|__|^|

PURSE: 45
Enter 1 to buy a Slingshot: DMG: 1 DELAY: 1 COST: 10
Enter 2 to buy a Catapult:  DMG: 5 DELAY: 3 COST: 20
Enter 3 to buy a Cannon:    DMG: 10 DELAY: 5 COST: 30
Enter 4 to continue.
```

This can continue until the user is ready to begin. When the option to continue is entered, enemies are randomly generated and placed at position 0 on the map. This is done by using a random number generator which will produce an int between 1 and 20.

(Whilst writing this bit of code and others I got a lot of checkstyle magic number warnings, which I understand are there to ensure that no numbers go unexplained. In this case however I feel like it makes my code a bit more complicated than necessary, but there you go.)

If the number is between 1 and 4, an enemy will be generated (1,2 gives a Rat, 3 a Camel, and 4, an Elephant). So, to start off with, there is a roughly 1/5 chance of an enemy being generated, with rats being twice as likely as the others. This changes as the game goes on - so for every ten time steps, the upper bound of the random number generator is decremented, making it more likely that an enemy will be generated.

There are 5 lines on the map that can hold enemies, so we loop through the random generator five times - once for each line. Back in the main method, we then check to make sure that there aren't just one or two enemies looking nervous - this makes for a rather short game! If there are two or less enemies, some more are added.

The game then loops through various methods to do the following:

1. Advancing each enemy according to their attributes (so Elephants move 1 step every 2 turns, Camels 1 step per turn etc).

2. Checking to see if any towers may fire on any enemies and if so, updating the enemies' health in accordance - another way of implementing the fire() function would be to loop through the existing enemies in the enemies array list and find the closest one in range, then shoot it. I tried this but it made the game very predictable, so I have opted to have the enemy that is fired on picked at random (as long as it is within range). This seems to me to better replicate the fog of war and makes for more tense moments - imaging a plucky rat zooming determinedly towards your battlements as cannonballs fly overhead. (Rather than being mercilessly and predictably slingshotted every single time.)

3. Randomly generating new enemies, as described above.

4. Giving the user the option to buy more towers or continue.

5. Updating the map with new, dead or advancing enemies and any new towers.

Because of the randomness of the enemy generation, some games are very hard, some are very easy, and most are in between. Here is an example mid-game. As you can see, the user is kept abreast of battlefield developments and the health of the enemies.

```
  0    1    2    3    4    5    6    7    8    9    10   11   12   13   14  |^|__|^|
                                                       _H\_ _H\_      _Y>_  |^|__|^|
/M@\ /M@\             ~>                                                   |^|__|^|
     /M@\                                                                  |^|__|^|
/M@\         MP                                                            |^|__|^|
             ~>                                                            |^|__|^|
       MP                                                                  |^|__|^|

Slingshot hit Elephant with 1 damage.
Catapult hit Camel with 5 damage.
Catapult hit Elephant with 5 damage.
Elephant HP: 10
Elephant HP: 5
Camel HP: 0
Camel died. You get 1 COIN for killing a sentient being!
Rat HP: 1
Camel HP: 5
Elephant HP: 9
Rat HP: 1
Camel HP: 4
Elephant HP: 9
PURSE: 6
Enter 1 to buy a Slingshot: DMG: 1 DELAY: 1 COST: 10
Enter 2 to buy a Catapult:  DMG: 5 DELAY: 3 COST: 20
Enter 3 to buy a Cannon:    DMG: 10 DELAY: 5 COST: 30
Enter 4 to continue.
```

The game is 'won' when the player has succeeded in killing all the animals in the corridor (but who really wins in war, anyway?) and lost if any animal reaches the end of the corridor and breaches the castle.

```
0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   |^|__|^|
                                                     _Y>_ _H\_ _H\_ _Y>_  |^|__|^|
                       ~>                                                 |^|__|^|
          /M@\                                                            |^|__|^|
            ~>                                                             MP _|^|
                       ~>                                                 |^|__|^|
          /M@\                                                            |^|__|^|

The animals have breached your defences. Fortunately, they're friendly and have brought cake.
```

As per spec, I have implemented the enemies and towers as subclasses of the Enemy and Tower class respectively. In addition to the methods specified, the Enemy class has additional methods such as setLine() and setPosition() which are used to set the enemy's location on the map. The tower class too has additional methods, including getCost(), getSymbol and more. (Please see Javadoc)

I have chosen to contain all the methods pertaining to the towers and enemies in their respective superclasses. An alternative way of doing this would be to have those same methods repeated in the different subclasses, and overridden them - but actually the only things that needed to change were the parameters passed to those methods. I gave each subclass of an Enemy or a Tower various final static attributes, such as their step, damage etc, which do not change throughout the course of the game. This enables the super constructor to be called with those variables, and removes the necessity of overriding the same methods in every class with only slightly different bodies, which seemed repetitive to me. I shall leave you with my UML class design for the game.