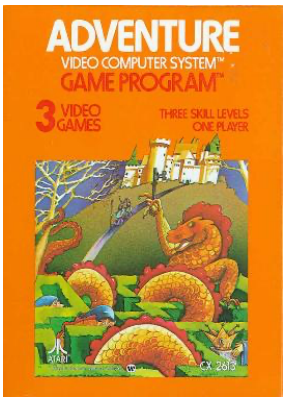
CSCI 1110: Assignment 2

Due: 11:59 pm, Sunday, October 23, 2022



The purpose of this assignment is to reinforce your understanding of object-based programming and to problem-solve using objects, classes, and nested loops. For this problem you will be provided with sample tests in Codio. All input is done via the console and all output is to be done to the console as well. You must submit your assignment via Codio, and you can test it by submitting your code. **Note: for this assignment you are expected to install and use an IDE (other than Codio) to develop your code. Please see How To videos in the How-To Tutorial Section of our Brightspace page on how to install an IDE**.

This assignment is divided into three problems, where each problem builds on the next. Each problem subsumes the previous one. Hence, if you complete Problem 3, you will have also completed Problem 2 and Problem 1.

Problem 1: A Player Tracker

Adventure games (named after the first such game[1]) involve the players going on a quest of exploration, looking for charms and completing a quest. Your team is working on a new adventure game and you have been asked to create the part of the game that tracks the players and the charms they have found.

Write a program called **`PlayerTracker.java`** that tracks the player's locations as they move through the game. Each player will have several properties, such as a name, the charms they have found, and their current location. Each charm will also have several properties such as an identifier, its location on the map, and what the charm does. **Note:** this program is very similar in some respects to Assignment 1.

Input

The input is divided into three parts: (i) a list of charms and their locations, (ii) the players being tracked, (iii) the tracked movements. The first part of the input is the list of charms. The first line contains a single integer denoting the number of different charms (***K***) in the game. This is followed by ***K*** lines encoding the information of each charm. Each line consists of the charm's identifier (***L***), a capital letter; followed by the charm's location ***X*** and ***Y***, the charm's effect ***E*** on a player's health, followed by the charm text. E.g.,

```
2
B 3 2 5 Blazing goblet
G 4 3 -10 Giggling lizard
```

In this example, the Blazing goblet improves a player's health by 5, but the Giggling lizard decreases it by 10. **Hint**: Use `nextLine()` to read the charm text **after** you have read in the identifier and location.

The next part of the input, is the players being tracked, consists of a single line containing an integer denoting the number of players being tracked (***P***), followed by ***P*** lines encoding the initial location of each player. Each line consists of their name and their location (**X**) and (**Y**). E.g.,

```
1
Alice 4 1
```

There are no limits on ***X*** and ***Y***. **Hint**: No need for a 2D array. Each player will have a unique name and each player's health is initialized to 100.

[1] Adventure, published by Atari, 1980 (https://en.wikipedia.org/wiki/Adventure_(1980_video_game))

The last part of the input, the movements, consists of a single line containing an integer denoting the number of movements (***M***), followed by ***M*** lines that encode movement information about players being tracked. Each movement record consists of a player's name (***N***), the change in their **X** coordinate and the change in their **Y** coordinate, e.g.,

```
4
Alice -1 0
Alice 0 1
Alice 1 1
Alice 1 0
```

Processing

Your program should track the player's movements and the charms they have discovered:

- Each player has a health score that is initially 100.
- Movement changes the player's location by adding the movement to the current location. For example, if Alice moves -1 1, this means that **X** is decreased by 1 and **Y** is increased by 1.
- After a player moves, their health decreases by 1.
- If a player has less than 1 health they cannot move and their moves are ignored.
- If a player arrives at a location with a charm, they collect the charm.
- If a player already has the charm, they do not collect it again. Charms are identified by their **id**.
- Collecting a charm involves removing the charm from a location.
- Collecting a charm increases or decreases the player's health, depending on the charm.

In Problem 1, assume that

- There is only one (1) player. I.e. ***P*** = 1
- The player has no charms initially.
- You **must** create a *Player* class and instantiate a *Player* object for each player
- You **must** create a *Charm* class and instantiate a *Charm* object for each charm.

Hints:

- Your *Player* class should use an array or *ArrayList* to store the list of charms collected by the player.
- **The basic algorithm is the same as for Assignment 1.** ☺

Output

The output consists of the status of each player. For each player output

- The player's name and location using the format: ***`Name (x,y) Health`***
- Followed by a list of charms they have collected. For each charm output "+" followed by the letter identifier of the charm, where the charm was located, and the charm itself. Use the format:

  `+ LetterID (x,y) Effect CharmText`

The output should be to the console, and each line terminated by a newline. The order of the players should be the same as the input order and the order of charms should be the order they were collected in. **Hint:** the `toString()` method of your Player class should build and return the entire string.

Examples

Input	Output
3 A 1 3 -7 Arboreal sock B 3 2 9 Blessed back scratcher G 4 3 -10 Giggling lizard 1 Alice 4 1 4 Alice -1 0 Alice 0 1 Alice 0 1 Alice 1 0	Alice (4,3) 95 + B (3,2) 9 Blessed back scratcher + G (4,3) -10 Giggling lizard

Approach

- You **MUST** implement the *Charm* class. You will need to determine which instance variables you will need. Implement the following **public** methods and constructors:

Constructor / Method	Description
`Charm(String id, int x, int y, int effect, String text)`	Constructor: stores the ***id***, ***x***, ***y***, ***effect***, and ***text*** in the *Charm* object.
`boolean isHere(int x, int y)`	Returns true if the specified location is the same as the Charm object's location.
`boolean isSameId(Charm charm)`	Returns true if the **id** is the same.
`int getEffect()`	Returns the effect of the charm.
`String toString()`	Returns a string representing the Charm object in the format described in the Output

- You **MUST** implement the *Player* class. You will need to determine which instance variables you will need. Implement the following **public** methods and constructors:

Constructor / Method	Description
`Player(String name, int x, int y)`	Constructor: stores the **name**, ***x***, and ***y***, in *Player* object.
`void move(int dx, int dy)`	Takes the change in position (**dx** and **dy**) of the player and adds it to the player's current location.
`int getHealth()`	Returns the current health of a player.
`void adjustHealth(int diff)`	Adds the ***diff*** to the player's health.
`boolean inSameLocation(Charm charm)`	Returns true if the *Charm* object is at the same location as the *Player* object.
`boolean addCharm(Charm charm)`	If the *Player* does not already have this charm, add it to the *Player's* collection. Returns true if and only if the charm was added.
`String toString()`	Returns a string representing the *Player* object, including all the charms, in the format described in the Output

- Implement Problem 1 of PlayerTracker.java. All you need can be done using the methods from the above classes. The algorithm is nearly identical to that of Assignment 1.
- **Hint:** You will need to use an array or *ArrayList* to store the *Charm* objects in your program.

Problem 2: Multiple Players

Extend your Player Tracker from Problem 1 to handle one change. There can now be multiple players tracked. I.e. ***P*** may be greater than 1.

Input

The input format is the same as in Problem 1.

Processing

Your program should perform the same task as in Problem 1 with the following assumptions

- Your program must handle tracking more than one player.
- Charms of the same type can be found at different locations.

Hint: You will likely need to use an array or *ArrayList* to store the *Player* objects.

Output

The output format is the same as in Problem 1.

Examples

Input	Output
4 A 1 3 17 Arboreal absinthe B 2 3 9 Blessed back scratcher B 3 2 9 Blessed back scratcher G 4 3 -10 Giggling Lizard 2 Alice 4 1 Bob 2 2 10 Alice -1 0 Alice 0 1 Bob 0 1 Alice 0 1 Bob 1 0 Alice 1 0 Bob 1 0 Alice -1 0 Bob -1 0 Alice -1 0	Alice (2,3) 93 + B (3,2) 9 Blessed back scratcher + G (4,3) -10 Giggling Lizard Bob (3,3) 105 + B (2,3) 9 Blessed back scratcher

Approach

- You **must** expand the *Player* class by adding the following **public** method:

Method	Description
`String getName()`	Returns the name of the *Player* object.

- Implement Problem 2 in `PlayerTracker.java`. All you need from the above classes is done. The algorithm is nearly the same as for Assignment 1.
- **Hint:** You will need to use an array or *ArrayList* to store the *Player* objects in your program.

Problem 3: The Full Player Tracker

Extend your program from Problem 3 to handle one more change. Players can effect other players with their charms.

Input

The input format is the same as in Problem 1.

Processing

Your program should perform the same task as in Problem 2 with the following assumptions

- Multiple players may occupy the same location
- When a player enters a location with other players, they apply the effects of their charms to all the other players.
    - The health of each of the other players is adjusted by the sum total of the effects of the incoming player's charms.
    - If the other player has the same charm, they are not affected by the charm.

Output

The output format is the same as in Problem 1.

Examples

Input	Output
4 A 1 3 17 Arboreal absinthe B 2 3 9 Blessed back scratcher B 3 2 9 Blessed back scratcher G 4 3 -10 Giggling Lizard 2 Alice 4 1 Bob 2 2 9 Alice -1 0 Alice 0 1 Bob 0 1 Alice 0 1 Bob 1 0 Alice 1 0 Alice -1 0 Bob -1 0 Alice -1 0	Alice (2,3) 93 + B (3,2) 9 Blessed back scratcher + G (4,3) -10 Giggling Lizard Bob (2,3) 86 + B (2,3) 9 Blessed back scratcher

Approach

- Expand the *Player* class by adding the following **public** methods:

Method	Description
`boolean inSameLocation(Player player)`	Returns true if ***player*** is at the same location as this *Player* object.
`void charmPlayer(Player player)`	Apply charms to **player** by adjusting their health

- Implement Problem 3 in `PlayerTracker.java`. All you need from the above objects is done.
- Iterate over all players to check who is in the same location and have the player who just moved give their charms to all the players in the same location and vice-versa.

What to Hand In

This assignment must be submitted in Codio via the Brightspace page.

Grading

The assignment will be graded based on three criteria:

Functionality: "Does it work according to specifications?". This is determined in an automated fashion by running your program on a number of inputs and ensuring that the outputs match the expected outputs. The score is determined based on the number of tests that your program passes. So, if your program passes t/T tests, you will receive that proportion of the marks.

Completeness of Solution: "Is the implementation complete?" This considers whether the classes nad methods have been implemented as specified. This is determined by visual inspection of the code. It is possible to get a good grade on this part even if you have bugs that cause the methods to fail some of the tests.

Quality of Solution: "Is it a good solution?" This considers whether the approach and algorithm in your solution is correct. This is determined by visual inspection of the code. It is possible to get a good grade on this part even if you have bugs that cause your code to fail some of the tests.

Code Clarity: "Is it well written?" This considers whether the solution is properly formatted, well documented, and follows coding style guidelines. A single overall mark will be assigned for clarity. Please see the Java Style Guide in the Assignment section of the course in Brightspace.

If your program does not compile, it is considered non-functional and of extremely poor quality, meaning you will receive 0 for the solution.

The following grading scheme will be used:

Task	100%	80%	60%	40%	20%	0%
Functionality (20 marks)	Equal to the number of tests passed.					
Completeness (10 marks)	Classes completed for Problem 3.	Classes completed for Problem 2	Classes completed for Problem 1	Classes are more than half complete for Problem I. (More than half the methods are done.)	Classes are less than half incomplete. Less than half of the methods are done	No code submitted or code is not a reasonable attempt
Solution Quality (10 marks)	Approach and algorithm are appropriate to meet requirements for Problem 3	Approach and algorithm are appropriate to meet requirements for Problem 2, but not Problem 3	Approach and algorithm are appropriate to meet requirements for Problem 1 but not Problem 2	Approach and algorithm will meet some of the requirements of Problem 1	Approach is correct but algorithm is incorrect	
Code Clarity (10 marks)	As outlined below					

Code Clarity (out of 10)

- 2 marks for identification block at the top of code
- 2 marks for appropriate indentation when necessary
- 2 marks for use of blank lines to separate unrelated blocks of code
- 2 marks for comments throughout the code as needed
- 2 marks for consistent coding style throughout

Hints and Things to Remember

- You may need to use a couple 1D arrays or ArrayLists.
- You **must** create and use *Player* and *Charm* classes and objects or marks will be docked.
- All input will be correct. You do not need to handle incorrect input, for now.
- Please install and use an IDE to develop this assignment. You will only be able to submit via Codio.