COSC-112-S18
Annabelle Gary, Sylvia Frank, Amy Pass
Final Project Grade Argument

a-MAZE-ing Maze!!

Functionality: We are very robust to keyboard inputs during the game. We ran many, many tests to see what happens when you press random keys during the game, and did many test cases to make sure that our code responds to user inputs the way it says in our write-up. We also tested all of the different ways you can die (and made sure that you actually die). Finally, we tested all possible map and item configurations. For example, one issue we found is that when you press caps lock on accident (for example, if you were to sit on the keyboard and hit it), then none of the key commands work because the letters are capitalized. We fixed this so that they work even if caps locked is hit. There are no graphical glitches; it may seem as though sometimes if you're very close to the edge of a path/slightly off the path you should die and you don't—this is not a graphical glitch. It is like that intentionally so that you can go 1/3 of the marble's width off the path just to give the user some leeway. As long as the user runs the program, there is pretty much no way for the game not to work.

Design:
Although our design may seem overly-complicated at first, there are very good reasons for the way we designed our project. For example, it may seem that having top and bottom corners, along with straight and horizontals, is over-engineering a solution where just horizontal and straight paths would have been easier. However, having these many subclasses of the Path made it more simple and elegant in the long run to implement other things (such as the Bumpers item, and the checkDead() method). This would also make it easier in the future to implement other changes to the game. One way that we were particularly clever is how we use the nextFrame() method and FPS to make the game has "time" to it. For example, being able to pause the game when you press h (so that you can read the instructions whenever you want) is one of our favorite parts of the project. Another example of using time in a clever way is our implementation of trigger events; i.e. after periods of time, the amount and placement of items on the path changes and the path sometimes (depends on a rand) speeds up.

Creativity: Creativity is the part of our project that is a bit lacking. Although the main idea of a marble on a path has been done very many times, we tried to make up for this with our items/boosters. For example, adding the aliens and the ability to shoot them is different from most of the classic "marble-on-a-path" games. The combination and sheer number of different boosters make our game go beyond the simple idea of a marble following a path.

Sophistication: Our sophistication is probably the best part of our project. Our project was mostly definitely more than 18 times the effort of doing an average lab (see the amount of commits and the times that we made them on github: https://github.com/agary110/CS112-Final-Project). The part of our project that is the most sophisticated, however, is the many different pieces which all have to fit together well. In particular, our code works very well to not only fit many different types paths together, but also different screens consisting of paths together. This works seamlessly, and it would be easy for a user to add or take away screens or paths (as long as each screen starts and ends at the same place: the middle of the screen). Additionally, our update methods work very well together; they manage to update all of the other classes seamlessly in every frame of the game. This exemplifies the great interconnectedness of our classes.

Broadness:

We used 1, 2, 5, 6, and 7.

1: We imported java.lang.StringBuilder in our Item class. We used this in the ammo counter to hold a place value when the ammo count is less than 10 (i.e. the ammo counter shows 01 instead of 1).

2: We used subclasses to make different types of paths and items. This subclassing was essential to the functionality of our project; it is the basis for the entire code. It was so important because in other classes we referencing a path or item without having to know what type of path or item it was.

5: We use linked lists to keep track of paths and screens. It allows us to easily iterate through the paths in a screen, while also allowing us to easily add and remove paths or screens.

6: Our file input/output gives us a place to store a high score (we wouldn't be able to do this otherwise once you die). This gives a player incentive to keep playing our game; they want to beat the high score!

7: Randomization is very important to our project so that the game is different every time you play it. For example, you get a different path each time you play, and different items pop up that do different things each time you play. It keeps the game unique and interesting every time you play!

Code Quality:

Code Quality:

Our code is of the utmost quality. Throughout each file, we maintain uniform commenting to make the functionality of each class and method clear. For each class, we provide a brief description of its purpose and functionality before providing its member variables, constructors, and methods. For each method, we provide the method name and its functionality using the following format:

```
    //======================
    /** Method: methodName()
            Functionality: This is the purpose and functionality of the method. **/
    //======================
```

Throughout each file, we maintain uniform coding organization. The order of our code is uniform. In each class, we used the following order to organize our code: import libraries, a brief description of the functionality of the class, the member variables, the constructors, and finally the methods. Through each file, we maintain uniform coding style to increase code cleanliness. In each file, we use a series of commenting techniques ("//==========" and "/** Section Title/Comment **/) to group portions of our code together. For example, in the Path class, these commenting techniques group together the public Path(int enterX) and public Path(Path previous) constructors because they are both constructors. We also end each class with "/** END OF _______ CLASS **/" to prevent confusion when files contain multiple classes.

Product Quality:

While we don't have the fancy graphics as contemporary games, it has a particular aesthetic that brings user back to Brick Breaker or Atari days. The simplicity of the style makes it fun and easy to start playing, while getting more and more addicting the more you play.